

TRI FILE COPY

ESD-TR-71-344

ESD ACCESSION LIST
TRI Call No. 74 875
Copy No. 1 of 2 cys.

THE LAYOUT PROBLEM FOR GRAPHS

Martha Greenberg Dennis

ESD RECORD COPY
RETURN TO
SCIENTIFIC & TECHNICAL INFORMATION DIVISION
(TRI), Building 1210

August 1971

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

Sponsored by: Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

ARPA Order No. 952

Approved for public release;
distribution unlimited.

(Prepared under Contract No. F19628-68-C-0379 by Harvard University,
Cambridge, Massachusetts 02138.)



AD0734037

LEGAL NOTICE

When U.S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

THE LAYOUT PROBLEM FOR GRAPHS

Martha Greenberg Dennis

August 1971

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

Sponsored by: Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

ARPA Order No. 952

Approved for public release; distribution unlimited.

(Prepared under Contract No. F19628-68-C-0379 by Harvard University,
Cambridge, Massachusetts 02138.)



FOREWORD

This report presents the results of research conducted by Harvard University, Cambridge, Massachusetts in support of ARPA Order 952 under contract F19628-68-C-0379. Dr. John B. Goodenough (ESD/MCDT-1) was the ESD Project Monitor.

This technical report has been reviewed and is approved.



EDMUND P. GAINES, JR., Colonel, USAF
Director, Systems Design & Development
Deputy for Command & Management Systems

ABSTRACT

The layout problem for graphs, the problem of automatically generating a representation of a graph on a two-dimensional surface, has been of interest in specific applications for many years, although little work has been done on the general problem. In this thesis three approaches are taken towards solution of the problem. The first approach defines general layout qualities believed to be desirable. Means for measuring these qualities in layouts and algorithms for their realization are developed. A graph layout building and modification system is described which provides an experimental environment for such layout algorithms. The second approach considers layout from an application dependent point of view. A classification of layouts into types is developed according to application, and layout algorithms for each type are discussed. In this classification, a correlation is found between complexity of layout type and complexity of layout algorithm. An extension of the above graph layout building system is designed, which allows for inclusion of application dependent information in layout processing. The third approach, that of considering the layout of modifications of graphs, rather than layout of whole graphs, is briefly considered. It is concluded that this third approach is the least effective for finding solutions to the layout problem.

TABLE OF CONTENTS

	Page
CHAPTER 1: INTRODUCTION	1
CHAPTER 2: GENERAL LAYOUT CRITERIA	6
2.1 REGULARITY, DIRECTIONALITY AND SIMPLICITY	7
2.1.1 Regularity	7
2.1.2 Directionality	16
2.1.3 Simplicity	25
2.1.4 Concepts in Aesthetics and Perception . . .	28
2.2 MEASUREMENTS AND REALIZATION.	37
2.2.1 Measurements	38
2.2.1.1 Repetition	40
2.2.1.2 Symmetry	43
2.2.1.3 Other Qualities of Regularity . .	49
2.2.1.4 Directional Consistency	55
2.2.1.5 Other Qualities of Directionality	64
2.2.2 Realization	68
2.2.2.1 Repetition	73
2.2.2.2 Familiar Figures	78
2.2.2.3 Link Length Consistency	78
2.2.2.4 Fidelity	85
2.2.2.5 Directional Consistency	86

	Page
2.2.2.6 Minimum Number of Bends . . .	92
2.2.2.7 Minimum Number of Intersections	98
2.2.2.8 Minimum Link Length	106
2.2.2.9 Parallelism	106
2.2.2.10 Horizontal-Vertical Orientation	107
2.3 THE MOD SYSTEM	108
2.3.1 The Input System	109
2.3.2 The Framemaker System	121
2.3.3 The Output System	123
CHAPTER 3: APPLICATION DEPENDENT LAYOUT	143
3.1 APPLICATION DEPENDENCY - A JUSTIFICATION	144
3.2 APPLICATION DEPENDENT LAYOUT TYPES	147
3.2.1 Classification of Layout Types	147
3.2.2 Layout Types and Algorithms	153
3.2.2.1 Linear Layouts	153
3.2.2.2 Tree Layouts	155
3.2.2.3 Network Layouts	174
3.2.2.4 General Layouts	188
3.2.2.5 Ordered-Arc Layouts	196
3.2.2.6 Summary of Layout Types	219

	Page
3.3 A DESIGN FOR THE EXTENSION OF MOD.	221
CHAPTER 4: ANOTHER APPROACH TO LAYOUT	236
APPENDIX 1: AMBIT/G OUTPUT PROGRAM	243
APPENDIX 2: AN ALGORITHM FOR REPETITION	252
APPENDIX 3: AN ALGORITHM FOR LINK LENGTH CONSISTENCY	276
APPENDIX 4: AN ALGORITHM FOR PARALLELISM	298
APPENDIX 5: DETAILS OF THE MOD SYSTEM	306
REFERENCES	313

Chapter 1

INTRODUCTION*

The "layout problem," the problem of generating an arrangement of objects on a two dimensional surface, was mentioned by Sutherland (34) as one of ten unsolved problems in computer graphics. Solutions to the problem for some specific applications have been found, but the general problem still remains unsolved. This work discusses the layout problem for the graph theoretic type of graph which Berge (5) has defined.** Such graphs consist of elements called nodes (or vertices) which are connected to one another by links (or edges). We will use the terminology "node" and "link." Links will be denoted by single letters or numbers or by a parenthesized pair listing the nodes they connect, for example, (a,b). Nodes will be indicated by letters or numbers.

The layout problem is then, given a fixed graph theoretic graph, which by definition is only a structural entity and has no physical characteristics, automatically generate a two-dimensional representation or layout of this graph. Many layouts are possible for each graph. In this work we only consider layouts in which links are drawn as series of straight-line segments. Line segments of

*This paper has also been included in the publication series of the Harvard Center for Research in Computing Technology as Technical Report 1-71.

**Additional references on graph theory are: Busaker and Saaty (10), Liu (25), and Ore (31).

links (also referred to as link segments) will be denoted by single letters or numbers or by a bracketed pair listing the endpoints of the line segment, for example, $[a,b]$. A point at which two link segments of a link are connected will be referred to as a "bend point," or simply a "bend."

In this dissertation no general solution to the layout problem for graphs is given, although some generally desirable criteria are discussed, along with methods by which these criteria may be realized in layouts. Furthermore, we consider solutions to graph layout in several particular applications in which graphs are used and in which layouts are needed. Many questions are brought out in these discussions, which remain unanswered and which require further work.

As an historical note, the author first dealt with the layout problem while attempting to develop an output program for AMBIT/G data. AMBIT/G is a computer language, the data and program of which are in the form of directed graphs. Thus the output problem for AMBIT/G was to generate and display a layout of some portion of the graphical data.

Two basic questions arose in the development of the output program, the first of which was how much of the layout was to be prespecified by the user and how much was to be generated automatically. Secondly, in the case of automatic generation of layout,

what criteria should be used and how should these criteria be realized?

The AMBIT/G output program finally developed allows for both prespecified and automatic layout. Automatic layout is used as a default condition in the absence of layout prespecification. Prespecification is accomplished by the building of a cumbersome super-structure over the data to be displayed* (see Appendix 1 for details), and only relationships between node positions can be specified.** Links are always routed automatically.

The automatic portion of the output program uses very simple criteria, disregarding such information as the original layout of the data and the general properties of the data graph. First, all nodes are placed on a grid, and then links are routed. When positions are not prespecified, automatic node placement is performed

* Even without layout prespecification, some super-structure has to be built in order to specify the portion of the graph to be output. This makes the whole output process difficult to use. In fact, another easier means for output was devised by one of the implementers of AMBIT/G. This was "on-line" as opposed to the method discussed above. In the "on-line" method the user started a display by naming a node which then appeared on the screen. He simply indicated which node he wished to appear next by pointing to a link origin in a node on the screen, or by naming another node and indicating where it was to appear. This method avoids all the complications of automatic layout generation.

** For example, we may specify that one node is to be placed to the east of another node.

as follows: if the node to be placed, a, is linked to a node, b, already placed, node a is positioned on a grid point adjacent to node b and in the direction of the connecting link from b, which direction is fixed, given b and the connecting link. If this position was previously occupied, or if node a has no connections with nodes already placed, it is placed to the right of those nodes already placed. Thus, the first criterion for automatic node placement is node connectivity, and the second is geometric linearity of a sort.*

Links are routed equally as simply, with paths deviating from straight lines only to avoid nodes. No attempt is made to avoid intersections, or to provide regularity. Thus, this simple solution to the AMBIT/G output problem avoids some of the more important considerations of the layout problem.

In this dissertation we will discuss some of the more basic problems involved in automatic layout generation for graphs and some possible approaches towards solution. The second chapter examines the more general comprehension criteria for layout. An attempt is made to separate out those application independent qualities of graph layouts which make some layouts more readable than others. Relevant literature from the fields of perception and aesthetics is considered. Discussion of the measurement and realization

* In other words, the default condition that a node is placed to the right of those already placed.

of such qualities is given. Included in this chapter is a description of the MOD system, a graph layout input/output system designed for experimentation with these application independent criteria.

The third chapter deals with the layout problem from an application dependent point of view. A defense is given for the argument that the application dependent approach may be more practical than the approach of chapter 2 for graph layout generation in some cases. A means for classifying several commonly used layout types for the purposes of layout is given. These layout types are discussed along with some layout algorithms. The chapter concludes with a design for including layout type information and type dependent layout generation in a graph layout input/output system like MOD.

Finally, the fourth chapter briefly considers another approach to the layout problem, that of laying out graph modifications, rather than whole graphs. This approach was found to be less productive than those taken in chapters 2 and 3.

Chapter 2

GENERAL LAYOUT CRITERIA

In attempting to characterize what makes a layout of a graph a good representation of the graph we must consider those overall qualities of layout which add to the readability of the graph representation. Here, we make an attempt to separate out and categorize those qualities which are generally grouped together so as to obscure definition. The categorization is based on the functions these qualities perform. Three broad categories which seem to account for graph readability are regularity, directionality (of reading), and simplicity. However, it will be shown that the third category is somewhat dependent on the first two. This problem of characterizing these qualities has previously been considered in studies of aesthetics and of perception.

After defining the qualities for overall layout, the feasibility of their measurement and realization in layout must be considered. Some work has been done in the direction of realization in the MOD system.

Section 2.1 discusses regularity, directionality, and simplicity, and considers previous work on this topic. Section 2.2 discusses the measurability of these qualities and the possibility of their realization in layout generation, and section 2.3 reports on the

work done in the MOD system.

It must be emphasized that the material of section 2.1 is a first attempt at classification of the layout characteristics which add to readability. This, by nature, is a subjective topic. The reader may disagree with the effectiveness and classification of some of the qualities discussed. He may also think of other qualities which should be included. What we aim at here is to provide a framework in which we can name and classify graph layout qualities, and understand and measure their effects. The contents of section 2.1, then, is only a beginning in the direction of this goal, and is based to some extent on the subjective views of the author, and those in her proximity.

2.1 REGULARITY, DIRECTIONALITY AND SIMPLICITY

2.1.1 Regularity

The category of regularity is a broad one. It includes those characteristics of layout which involve repetition, consistency, and the occurrence of easily recognized geometrical forms. Regularity seems to be one of the most important qualities responsible for making graph representations or layouts readable. Certainly, large layouts must be read in subsections, and where subsections are similar, the layout is more easily subdivided by the eye. Furthermore, the fewer the types of subsections there are to comprehend,

the easier the pattern matching necessary for reading. It is clear then that the larger the layout, the more important regularity becomes for comprehension.

Let us consider the most obvious type of regularity, repetition. There are several forms repetition may take. We first name and define these forms:

1) Literal repetition: two subparts of the layout^{*} are congruent and have the same orientation (also referred to as identical repetition).

2) Symmetrical repetition: two subparts of the layout are reflections of one another, with respect to an axis; i. e., if one of the subparts is flipped over this axis, it will lie on top of the other subpart.^{**}

3) Rotational repetition: two subparts of the layout are congruent but have different orientations, or, the mirror image of one subpart is congruent to another subpart, but the two subparts do not fulfill the requirements of symmetrical repetition; i. e., their orientations are different.

^{*} It is assumed here that the subparts referred to here and below lie in different positions.

^{**} Point symmetry, in other words, symmetry with respect to a point (such as that found in a pin wheel) is not considered here.

4) Similar repetition: two subparts are geometrically similar to one another but not congruent, or, the mirror image of one subpart is geometrically similar, but not congruent, to another subpart.

Let us first consider literal repetition. When a subpart of a layout is repeated exactly one or more times, it tends to identify itself as a distinct subpart of the layout, lending to the viewer's ability to subdivide the layout. But the number of these repetitions which may exist in a good layout is limited. For example, quickly glance at figure 2-1a. The structure is quite clear. Now do the same for figure 2-1b. How many nodes are there? In 2-1b the number of repetitions, it seems, are too many to comprehend at once, whereas in 2-1a we can easily do this. Now if we modify 2-1b slightly to obtain 2-1c, we notice that the result is much easier to read. The repetition is on two levels; we read the layout as three

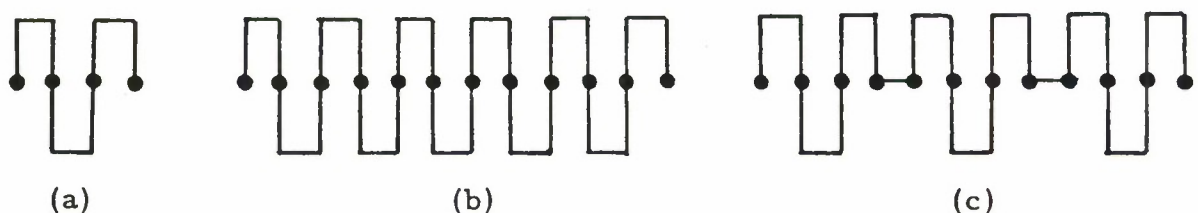


Figure 2-1

units of four nodes each. We are again reading the layout using literal repetition, but more easily than in 2-1b, since the nodes are divided into countable units. Thus repetition may occur at many

levels in a layout, and the layout remains readable as long as the number of repetitions of a given unit at a given level remains easily countable.

We notice that repetition may describe the relationships of non-distinct subparts of a layout as well as those of distinct subparts. For example, we understand the layout of figure 2-2 easily because we see four identical, but non-distinct subparts.

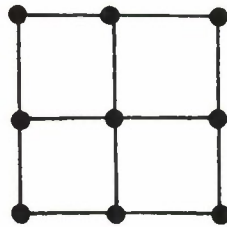


Figure 2-2

Symmetrical repetition also acts as an aid to comprehension of layouts. For example, in figure 2-3a, the right side is a reflection of the left with respect to a vertical axis, adding to readability, whereas in 2-3b, there is no symmetry.

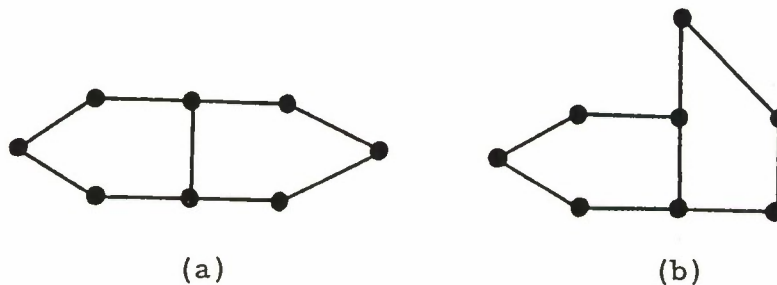


Figure 2-3

There is some question as to whether rotational repetition aids significantly in layout readability. It is quite clear that rotational repetition is not as effective as literal or symmetrical repetition. How much clearer figure 2-4a is than figure 2-4b seems mainly to depend on the particular viewer's ability to identify rotational instances.



Figure 2-4

There is also some question about the advantage of similar repetition. One factor which seems to determine the ease with which one detects similar instances in a layout is the closeness in size between the two instances. For example, parts (i) and (ii) in figure 2-5a are seen as similar much more easily than parts (i) and (ii) in figure 2-5b. This phenomenon may be related to that of size consistency, which is discussed below. Orientation also seems to contribute greatly to the detection of similarity. When the similar instances are oriented identically, or when the reflection of one instance with respect to some axis is oriented identically to the other instance, the two similar instances seem much easier to associate

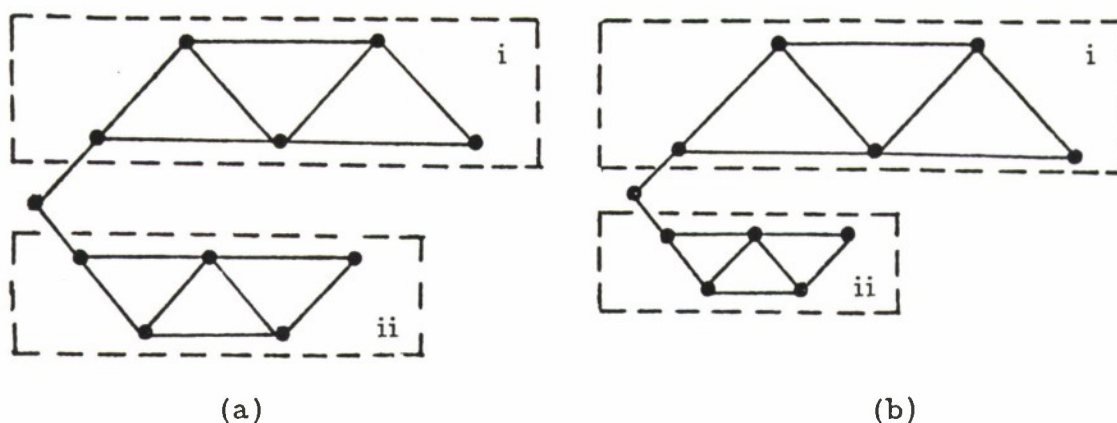


Figure 2-5

than two similar instances not oriented in these ways. Examine, for example, the pairs of triangles in 2-6a and b as opposed to the pairs in 2-6c and d.

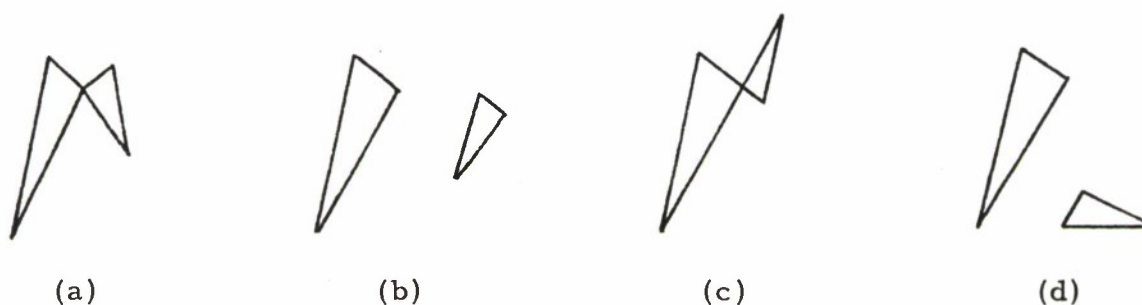


Figure 2-6

We must also consider as part of regularity, the occurrence of certain very familiar, easily recognized geometrical patterns such as lines, triangles, squares, regular polygons, etc. These familiar aids help the viewer to organize a layout, since they are easily seen when they occur, and thus lead to a faster comprehension of a layout.

A main component of regularity which seems to underlie much

of what was mentioned above is what we shall call size and distance consistency.* In general, this implies equivalence of link lengths, where possible, and, at a more subtle level, what Baecker (4) calls fidelity, which will be described below. Applying the equivalence criterion where possible, similar images tend to become closer in size, some literal, symmetrical, and rotational repetitions tend to appear, and in general the graph layout becomes more regular. This explains the discussion of figure 2-5. Application of this constraint in its literal form to the layout of figure 2-7a might result in the layout of figure 2-7b, a definite improvement. This length constraint, which we will call "link length consistency," will be taken to mean that as few different link lengths as possible appear in the layout.



Figure 2-7

*The concept of size and distance consistency is intended to apply to layouts in which links consist of a single line segment. The extension for multi-segment links, although not considered here, might be worthwhile to explore.

Baecker has expressed size and distance consistency in another form, which he calls fidelity. He suggests that graph layouts are better, the more the graph-theoretic distances and layout distances between nodes correspond. In this sense he is concerned with how faithful a representation of a graph is to the graph itself. For example, in figure 2-8a, node a is separated from node b by at least two links (hence, the graph-theoretic distance is two), and yet it is drawn closer to node b than to node c, which is adjacent to a. In figure 2-8b the layout has better fidelity.

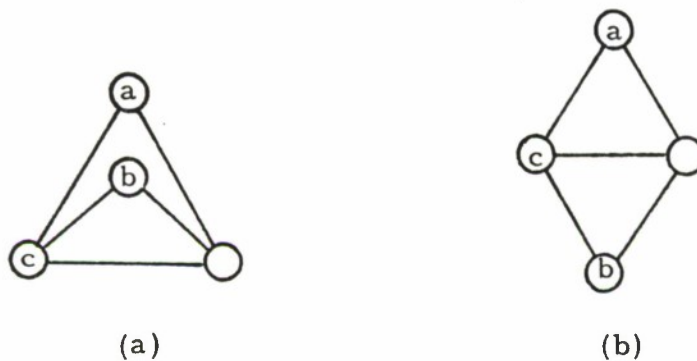


Figure 2-8

It seems appropriate to discuss one final layout characteristic, balance, under the topic of regularity. A crude definition of balance might be evenness of node and link distribution throughout a graph layout. We want to separate symmetry from balance, however, so as not to attribute clarity to balance when it is due to symmetry. Let us look at the graph layouts in figure 2-9a and b. The

amount of symmetry seems the same in both, but b is much better balanced than a, in other words, the nodes and links are distributed more evenly. There is some question as to whether balance contributes significantly to readability, or whether it just adds to the aesthetics of a layout.



Figure 2-9

In summary, we have included under the category of regularity several layout qualities, all of which contribute to the readability of graph layout by adding consistency or by facilitating some form of pattern recognition. This category includes literal, symmetrical, rotational, and similar repetition of both distinct and non-distinct layout subparts, which may occur on many levels. The appearance of familiar figures was considered, as well as consistency of size and distance in layout. Finally, balance was discussed as a possible aid to readability.

2.1.2 Directionality

Another category of qualities, which we shall call directionality, deals with the layout as a whole rather than in terms of its parts. Directionality includes those qualities which aid in a directed reading of a layout. This may be taken literally, or may be considered at a more subtle level. To some extent, then, directionality seems to deal more with links and their paths in a layout, rather than with nodes.

What first comes to mind is the nature of the links in a layout. If the layout is directed* is there some consistency in the direction in which the arrows point? Flow-type and network diagrams by convention ask that there be some directional consistency, often requiring that arrows point predominantly in one direction, say to the right. The constraint of directional consistency may also be met when arrows point consistently toward, away from, or around a center, as in figure 2-10. In general, directional consistency for directed layout means that there is some regular manner in which arrows are arranged. It is clear that this quality aids in the ease of reading directed layouts by providing some pattern in eye movement.

Two other qualities seem to be good candidates for

* In other words, links have a direction, indicated by arrows. This type of graph representation is used when the underlying graph is directed.

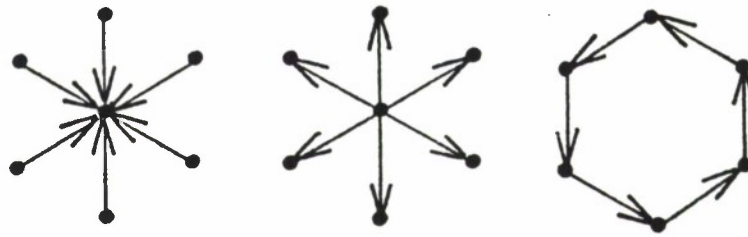


Figure 2-10

directionality. These are, first, the number of bends, or equivalently, link segments used in representing the links of the graph, and, second, the total link length used in representing these links. We will call these qualities "number of bends" and "total link length," respectively. Both qualities have been considered as layout criteria in circuit layout. In the first case, clearly, the fewer the bends in a representation of a link, the easier it is for the eye to follow its path. Furthermore, in the places where bends occur, in some cases, there is a tendency for the eye to create nodes, for example, in figure 2-11.

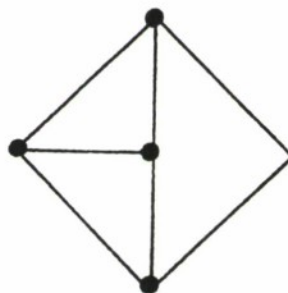


Figure 2-11

In the second case, it seems obvious that the greater the total link length for a layout, the more complex the layout will be. First of all, we find that the more link length there is, in general, the greater the probability that bends will also be present. Furthermore, when link length is in excess in a layout, it is usually because some other criterion such as the number of intersections is being minimized in the layout. Thus the complexity due to large link length in a layout may really be due to inherent graph complexity in the underlying graph. To illustrate, consider figure 2-12a. We may attribute its complexity to link length (and to the number of bends). However, we see in figure 2-12b, the same graph represented with a smaller total link length contains other features which reflect the inherent complexity of the underlying graph.

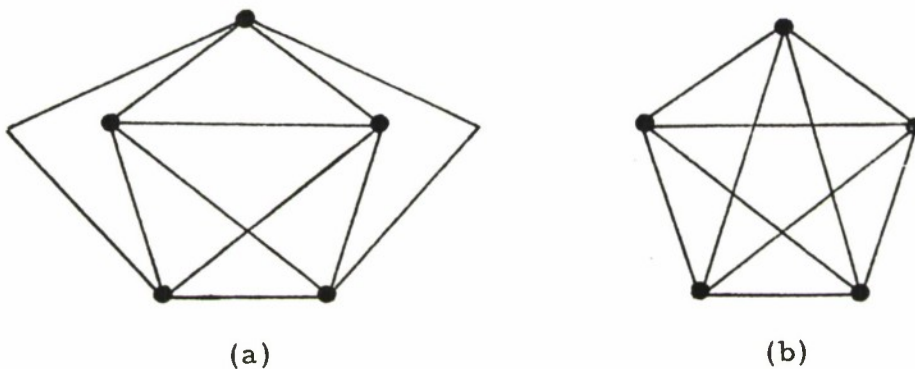


Figure 2-12

Another quality which seems to affect directionality in a layout is the number of link intersections other than at nodes. As is well

known in graph theory, some graphs cannot be drawn on a two-dimensional surface without intersections. These are called non-planar graphs. In fact, the minimum number of intersections which can be attained for any two-dimensional representation of a given graph is inherent in the graph and is called the genus of the graph.* For example, the graphs represented in figure 2-13 are non-planar.

We might require for readability that the number of intersections in a layout be as small as possible for the underlying graph. For instance, the minimum number for the graph represented in figure 2-14 is zero, and we find that 2-14a is easier to comprehend than 2-14b, where the intersection adds complexity similar to that added by bends in links. A further complication produced by



Figure 2-13

* With this quality, one of the main problems underlying the layout problem surfaces. This is the problem of differentiating what part of the layout is determined by the graph theoretic structure of the underlying graph and what part depends only on a Euclidean metric. In the case of intersections the effects of graph theoretic structure are quite clear, but in most cases, the effects are not as clear.



Figure 2-14

intersections is that faces are created visually which are not true graph theoretic faces. This also leads to difficulty in comprehension.

But consider the layouts in figure 2-15. The number of intersections in 2-15a is much greater than in 2-15b, yet we do not find it particularly more complex. The same situation occurs in comparing 2-15b to 2-15c. Here we have a case where the intersections, due to their number and regularity, do not detract from the directionality of the layout. Thus the importance of the minimum intersection constraint is more difficult to assess than was first expected. We must also take into account how much the particular intersections add or detract from directionality.

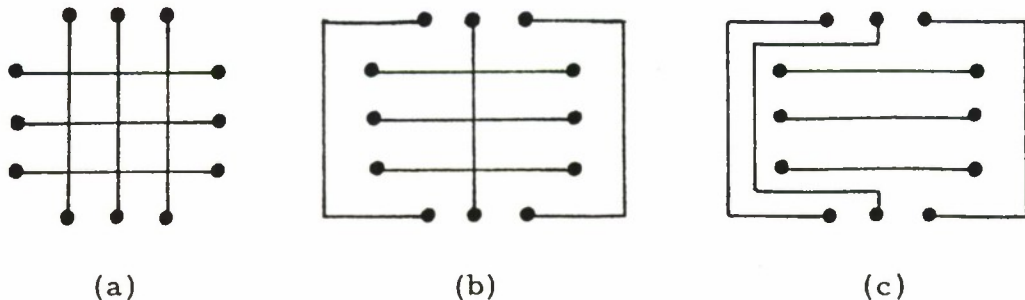


Figure 2-15

We will now consider a more subtle aspect of directionality, that involving link (or line segment) orientation. Two questions arise when we consider orientation. First, what effect does the variation of the slope of the link segments have on the layout? And second, is a generally horizontal-vertical link segment orientation better than any other general orientation?

To answer the first question, examine figure 2-16. The layout in figure 2-16a is quite clearly better than that of figure 2-16b. There are several factors which contribute to this difference. In one



Figure 2-16

figure, the number of parallel lines is large, the number of different slopes is small, and angles between link segments are limited to 0° and 90° . In the second figure this is not the case. Let us consider each of these factors separately.

We first conjecture that parallel lines are much more effective in layout than non-parallel lines. The larger the number of different link slopes in a layout the more confusing it is. Examine

the gradation of the layouts in figure 2-17, for example. Recurrence of a particular slope in the form of several parallel lines tends to reinforce a direction in the layout. Whereas, lack of such reinforcement may generate directional confusion. This quality will be referred to as "number of link slopes," or, equivalently, "amount of parallelism."

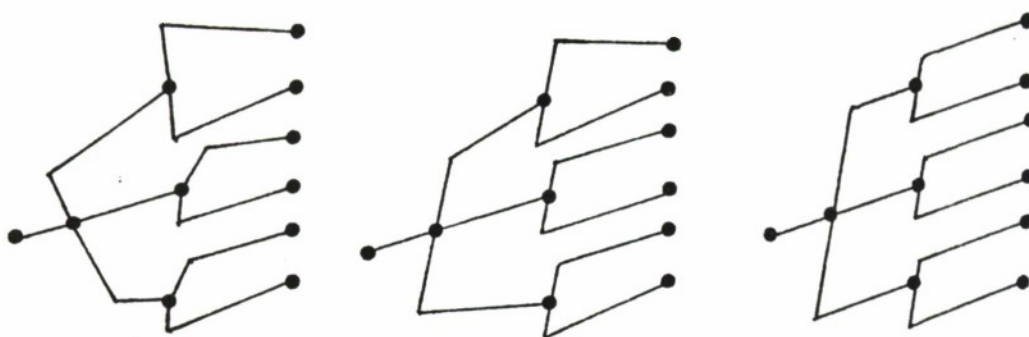


Figure 2-17

We then ask whether certain angles between links are preferable to others. One might first guess that 90° is a preferable angle. However, in examining figure 2-18, we see that this is not necessarily the case. This first guess, we find, is instead accounted for

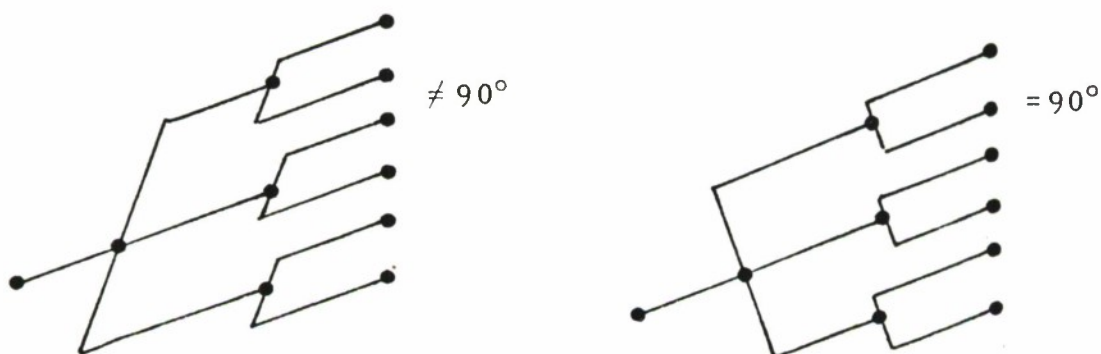


Figure 2-18

when we consider horizontal-vertical link segment orientation.

This lack of preference may be explained somewhat by our ability and tendency to see things in perspective, and to give a three-dimensional reading to a two-dimensional figure.

We will now consider the question of whether a horizontal-vertical link segment orientation is preferable to any other. By horizontal-vertical link segment orientation, more specifically, we mean that the link segments of a layout are oriented either horizontally or vertically. The answer to the question of whether such an orientation is preferable is obvious in the comparison of figures 2-19a and b. The reason for this preference is not clear, although it has been mentioned as an important factor by Birkhoff (6).*

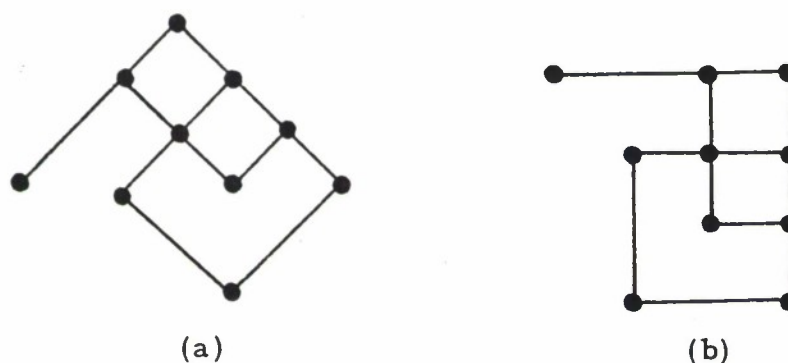


Figure 2-19

*Actually, Birkhoff mentions two factors which might contribute to this preference. One is what he calls "equilibrium," which accounts for whether a figure looks stable or unstable. The other factor, which he calls "the relation to a horizontal-vertical network," expresses the fact that humans prefer figures whose sides (link segments) fall on a horizontal-vertical network, or, as a second preference, on a network whose grid cells are diamonds. Birkhoff relates this factor to our everyday experience.

Thus, if we were to pose the question about angle preference with the example in figure 2-20, we might arrive at a different answer. This can easily be explained by preference for horizontal-vertical orientation.

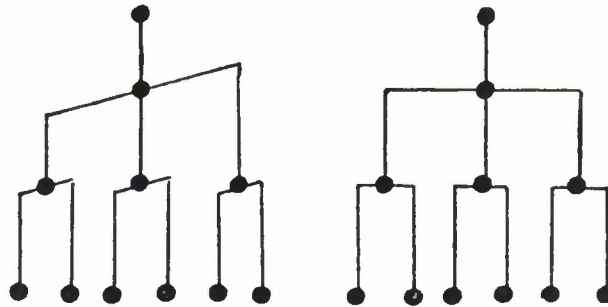


Figure 2-20

In summary, we have considered several layout qualities under the category of directionality. These have the common feature that they all contribute to some directional organization of the layout of a graph. The most obvious quality which has been considered is directional consistency in directed layouts. Bends in links, total link length, and link intersections have also been examined for their effect on directionality. Aspects of link (or line segment) orientation in a layout have been discussed; this includes the qualities of parallelism, number of different link slopes, and size of angles between links in a layout. Finally, horizontal-vertical link segment orientation preference has been considered.

2.1.3 Simplicity

It seems appropriate that we should also consider simplicity as a category of layout qualities, since most observers tend initially to equate clarity and simplicity to some degree. Let us try to define what is meant by simplicity in a layout. The most obvious definition is that a layout is simplest when it represents the underlying graph^{*} in the most straightforward manner possible, and it avoids unnecessary complication. For example, figure 2-21a is certainly simpler than figure 2-21b, because of the unnecessary complication of a link intersection in 2-21b.

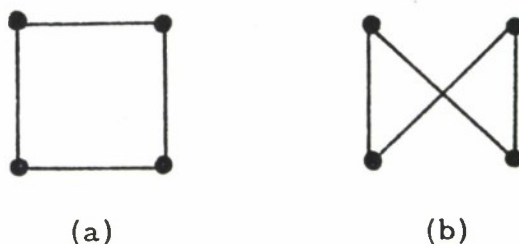


Figure 2-21

We must then ask what qualities tend to add to the unnecessary complication of a layout. And, are these qualities different from those discussed under regularity and directionality.

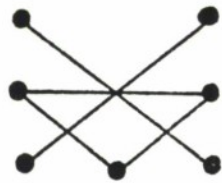
To review, the qualities included under regularity are those which affect the reading of a layout through its subparts.

^{*}Note that this issue is not to be confused with the graph theoretic simplicity (or complexity) of the underlying graph itself.

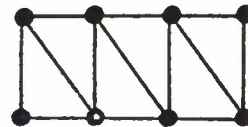
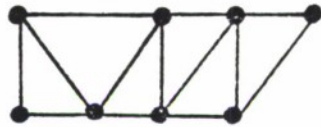
Directionality, on the other hand, includes those qualities which affect the overall direction and orientation of a layout. As can be seen in many of the examples above, optimizing on the qualities discussed under these categories seems to produce simpler layouts. Unnecessary complication can usually be pin-pointed as a lack of optimization of one or more of these qualities. It seems, furthermore, that with the categorization given above, any quality which affects the simplicity of a layout, is more specifically affecting either the regularity or the directionality of the layout and should be categorized accordingly.

From this discussion, we conclude that simplicity, in itself, should not be considered as a separate category, but as a complex of the effects of qualities we've already considered under regularity and directionality.

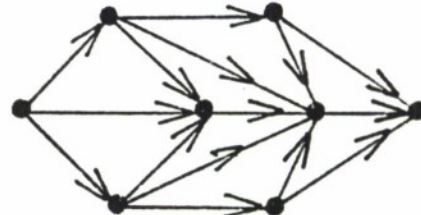
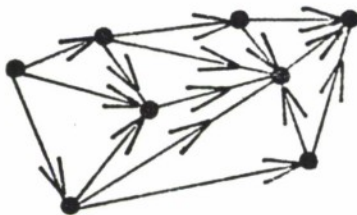
Some further examples may help convince us of this point. Consider the pairs in figure 2-22. In each pair there is one layout which is obviously simpler than the other. The difference in each case can be understood by pointing to a quality or set of qualities discussed under regularity or directionality which accounts for the simplicity or complication present. In figure 2-22a the most obvious quality is the number of intersections. In b, it is the existence of literal repetition. In c both symmetrical repetition and directional consistency contribute, whereas in d we find that parallelism and



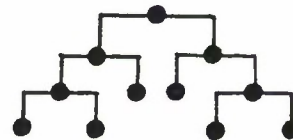
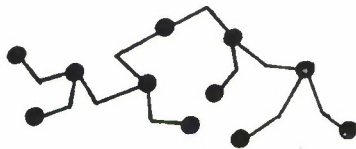
(a)



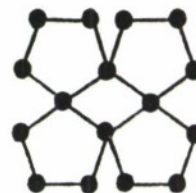
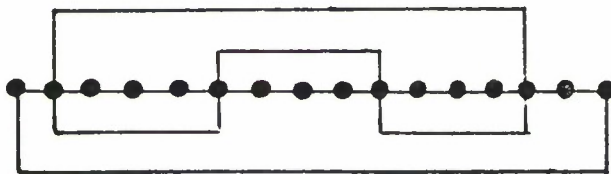
(b)



(c)



(d)



(e)

Figure 2-22

horizontal-vertical link segment orientation accounts for most of the difference. Finally, in e the separation into literally repetitious subparts as well as symmetrical repetition makes one figure simpler than the other. The reader should try several pairs himself, and examine the qualities which account for simplicity or complication. Perhaps, in this examination, other qualities which have not been brought out above may appear.

2.1.4 Concepts in Aesthetics and Perception

In this section we will supplement the discussion of sections 2.1.1 through 2.1.3 by considering some relevant ideas found in the literature of aesthetics and psychology. Both fields have dealt with the question of what factors add to the readability of a two dimensional layout, or, more generally, a two dimensional figure.

In studies of aesthetics, the motivation for answering this question is to understand what factors add to the aesthetic quality of a figure or an object. And as Birkhoff reports, Helmholtz stated: "The more easily we perceive the order which characterizes the objects contemplated, the more simple and perfect will they appear, and the more easily and joyfully shall we acknowledge them" (6, page 199).

In psychology, an understanding of the factors contributing to readability of figures is linked to an understanding of human visual

perception.

In the remainder of this section we will briefly describe some of the relevant literature in these two fields. In doing so, we will try to list those factors thought by the authors to contribute to readability, and briefly examine how these factors correlate with the ideas given in the previous three sections.

Let us start with aesthetics. The wish to understand what factors cause a sensation of aesthetic feeling when an object is perceived, has long been of concern to philosophers. Few, however, have attempted to describe aesthetic factors in a formal manner. The one exception seems to be the careful work done by George Birkhoff in 1933 on this subject.

To Birkhoff, "the fundamental problem of aesthetics [is] to determine, within each class of aesthetic objects those specific attributes upon which the aesthetic value depends" (6, page 3). In summary, Birkhoff first proposes that the measure of the aesthetic value (M) of an object may be determined by the complexity (C) of the object, and the order (O) or harmony of the object, according to the formula: $M = O/C$. The complexity (C) of an object is a measure of the amount of effort which must be expended in perceiving the object. The measurement of complexity is different for each class of objects.

The order (O) of an object is determined as a summation of the measures of various "elements of order" for the object. The elements

of order reflect the various attributes of the object contributing to or detracting from the aesthetic value. Positive elements of order are, for example, repetition, similarity, contrast, equality, symmetry, balance, and sequence (6, page 9). Negative elements of order include ambiguity, undue repetition, and unnecessary imperfection (6, page 10). Birkhoff determines for each class of objects those elements of order which he feels contribute to the measure of order (O).

Having established this formalism, Birkhoff proceeds to derive measures of aesthetic value (M), for several classes of objects found in art, music, and poetry. For each such class, he determines the set of elements of order to be considered in the measure of order (O) for the objects in this class, as well as some measure of the complexity (C) of these objects.

Of particular interest to us is his study of the class of polygons, in that polygons are so integral a part of graph layouts in which links consist of straight line segments. In the class of polygons, the elements of order Birkhoff considers are briefly described below:

- 1) Vertical symmetry: the figure is symmetrical with respect to a vertical axis.

- 2) Equilibrium: the figure rests on a horizontal base with optical center of gravity above this base.

3) Rotational symmetry: the figure is symmetric with respect to a point at its center.

4) Relation to a horizontal-vertical network: the sides of the figure lie along lines of a horizontal-vertical network.

5) Unsatisfactory form: Birkhoff's "omnium gatherum" of negative factors.

Given a figure, for each of these elements of order, i , a quantity, x_i , is determined, according to how the figure meets the requirements of the particular element of order. The order (O) is then the sum of the measures x_i . The details of how a given x_i is calculated will not be mentioned here.

Birkhoff then proceeds to calculate the aesthetic value (M) for a large group of polygons. According to his analysis, the square is the most aesthetic of all polygons.

Let us now briefly compare Birkhoff's selection of aesthetic factors with the group of readability qualities discussed in sections 2.1.1 and 2.1.2. The idea basic to both approaches is the concept of order; underlying this we find that both approaches consider symmetry (although Birkhoff includes point symmetry), horizontal-vertical orientation, diversity of directions, and similarity within figures (a concept which Birkhoff discusses when he considers ornaments as a class of objects to be analyzed).

In addition, several factors not considered in sections 2.1.1 and 2.1.2 are brought out by Birkhoff. In this author's opinion, however, these factors tend to add more to aesthetics than to readability. Among these factors are Birkhoff's idea of equilibrium, which relates, in a sense, to the idea of balance (section 2.1.1), and the concepts of unnecessary imperfection and ambiguity.

We will now examine some of the ideas found in the literature of psychology, which are pertinent to the problem of readability. As mentioned above, the problem became important in this field when questions of human visual perception were seriously considered. The basis of many of the current concepts in visual perception was developed in the writings of the Gestalt school of psychologists. According to Boring (7), this school began in 1912 with the writing of Wertheimer which "treats of the general dynamics of the formation of form" (7, page 252). Several well-known publications cover the concepts of the Gestalt school, including those by Hartmann (18), Koffka (22), and Katz (21).

As a brief background note, the approach of the Gestalt psychologists differs from previous approaches to perceptual studies in that the Gestalt studies were basically phenomenological, rather than physiological. In other words, perceptual phenomena are "allowed to speak for themselves" (21, page 18), rather than being subject to various types of physiological analysis.

According to Boring, "the chief contribution of Gestalt psychology to the psychology of perceived form was its insistence that the perception is formed under certain dynamical laws which give it its specific psychological organization. A perception is not a copy of its stimulus" (7, page 246). And according to Allport (1), these "laws are natively given, and are a property of the organizing action of the nervous system" (1, page 115).

Allport states that "no less than '114 laws of gestalten' have been formulated by various writers" (1, page 113). In subsequent literature, the list has been made more compact. Most of these principles apply to visual form. Let us first summarize a few of the more general principles of Gestalt with relevant quotations from Allport (1, page 113):

1) Form-concept isomorphism: "When one perceives an object that object tends, psychologically, to take on form; and forms establish themselves and persist . . . such forms occur within the nervous system or brain as macroscopic states or physiological configurations which are isomorphic . . . with the configuration of the percept to which they give rise."

2) Wholeness-character and relationships: "The form always has a 'whole-character' that transcends the characteristics of the parts. . . . The perceiving of relationships is an essential aspect of wholeness in experience."

Allport also includes other principles stating that laws "intrinsic to the organism" underlie the perception of form. The perceived configurations tend to be "self-closing and to be simple, balanced, and symmetrical. The tendency is toward 'good' form." In addition, the configuration is organized by certain forces, "giving rise to segregation, groupings, combinations into subsystems, and articulation."

Some of the more specific laws for producing visual form have been summarized by Katz. These specific laws are of more interest to us, in that they describe in more detail what factors help organize perception of a two-dimensional representation. A few of these are (21, page 25 ff.):

- 1) The law of proximity: "Other things being equal, in a total stimulus situation those elements which are closest to each other tend to form groups."
- 2) The law of similarity: "When more than one kind of element is present, those which are similar tend to form groups."
- 3) The law of closed forms: "Other things being equal, lines which enclose a surface tend to be seen as a unit."
- 4) The law of "good" contour, or common density: "Parts of a figure which have a 'good' contour, or common density tend to form units." (Also known as the law of "good" continuation.)

Wertheimer (38) and Koffka, who have originated much of this work, both discuss these laws in detail and give several examples of their application. In 1940 Mowatt (29) published the results of an experiment to test the value of these laws. In summary, her subjects were given a series of drawings and asked to change them in any way desired to produce what they felt were "good" figures from the drawings. She found that, in general, subjects changed drawings in such a way as to increase, among other things, differentiation, simplicity, closure, symmetry, good continuation, and occurrence of familiar forms.

We see then that some of the Gestalt concepts of visual perception agree with some of the factors mentioned in sections 2.1.1 and 2.1.2, for example, the importance of symmetry, similarity, and the occurrence of familiar forms. Other factors are mentioned by the Gestaltists which should be examined as possible additions to the list of readability qualities. Among these are the concepts of proximity, good continuation, and closure, none of which have been considered in the previous sections.

More recent work on the problem of determining the factors affecting visual perception has been done by Attneave and Arnoult (3). They mention that the problem underlying work in the area is that: "Our most precise knowledge of perception is in those areas which

have yielded to psychophysical analysis (e.g., the perception of size, color, and pitch), but there is virtually no psychophysics of shape or pattern" (3, page 123). And in regard to natural form, "relatively few scientists have seriously applied themselves to the problems of analyzing and describing form; these problems seem to have fallen into the cracks between sciences, and no general quantitative morphology has ever been developed" (3, page 132). They mention the work of Thompson (35) as the only major work in the field, but a work which is limited in its contribution to the identification of psychophysical variables of form.

Hake (17) has summarized some of the more recent experiments with the factors involved in visual perception. Among these he mentions experiments in judgement of complexity of figures, in which results show dependency on the number of turns or angles, and upon symmetry. He also summarizes experiments on the effects of redundancy in figures, and concludes that the helpfulness of redundancy in figures depends on the context of the figures. Experiments on the effects of rotation are also mentioned. In all rotation experiments reported on, accuracy of figure recognition is impaired by the rotation of a figure. Hake also reports on a large amount of experimentation with figures tilted toward or away from the observer. To some degree, it is found that an adjustment in the perception of such a tilted figure is made for perspective.

Comparing Hake's discussion with that of sections 2.1.1 and 2.1.2, we find support for the ideas found in the discussions on the number of bends, rotational repetition, and angles between slopes. However, some question about the effectiveness of symmetry and of literal repetition is raised by the experiments reported on by Hake.

As a final note, we mention the work of Miller (28), who has dealt with a problem implicitly mentioned in section 2.1.1, namely, the problem of what a "countable" number of repetitions is in a layout. In his essay, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," Miller presents evidence that humans seem to be limited in what he calls "channel capacity," in number estimation ability, and in immediate memory span, by a number in the range of seven. Although he draws no conclusions about this "coincidence," the results of several experiments are reported on. These experiments suggest that, perhaps, the answer to the problem posed in section 2.1.1 is seven plus or minus two.

2.2 MEASUREMENTS AND REALIZATION

Having established some of the qualities which we believe contribute to readable graph layouts, we would like to be able to measure how much a particular layout fulfills a particular quality. Furthermore, we wish to examine the possibilities for the

realization of these qualities. In other words, can we provide some method to optimize these qualities in layouts? Both problems are quite complex and require close examination.*

2.2.1 Measurements

The ideal measurement of a quality in a layout tells us to what extent we have optimized the layout for the particular quality. There are two types of measurements we would like to consider. The first, which we shall call the "normalized measurement," states the extent to which a quality is fulfilled in a layout, relative to the optimal possible fulfillment for any layout of the particular underlying graph. The second, which we shall call the "non-normalized measurement," does not take the optimal case into consideration. This second measurement gives us some idea about how well a quality is fulfilled in a layout, but is useful only when a comparison of two layouts of the same graph is made, and the best of the two is to be chosen. It does not tell the extent to which one layout is better than the other, with respect to a particular quality.

The normalized measurement of a quality requires that we have a method to establish the absolute minimum (or maximum) for

* Note that in this work we do not examine the question of internal representation of graphs and layouts although the nature of this representation has an effect on the efficiency of measurement and realization techniques, and should eventually be looked into.

this quality and a particular graph. It seems that for many qualities, this is a very difficult task. For example, with the minimum number of intersections, this would mean the establishment of the genus of the graph. Furthermore, the absolute optimum, given a graph, may be dependent to some extent on factors which are irrelevant to the particular quality under consideration. For example, the optimum may depend on other qualities to which we give priority in layout, if these qualities conflict with the particular quality under consideration. Then we must modify the definition of "absolute optimum given a graph" to "optimum given that the layout will be optimized first for qualities with higher priority." Thus, a more feasible approach to measurement is the use of the non-normalized measurement. In fact, in many optimization procedures, this is all that is necessary since only the relative maxima (or minima) are sought.

In the following discussion of specific measurements, we will only consider measurements for those qualities which we feel are most effective for layout clarity. Thus, measurements will not be examined for qualities defined in sections 2.1.1 and 2.1.2, which are not believed to contribute significantly to readability, such as rotational repetition, balance, and angles between links. The main aim is to find at least one non-normalized measure for each quality. Should a normalized measure be found easily for a quality, it will also be discussed. Even with the simplest measurements as our

goal, however, we find that there are some qualities which appear very difficult to quantify. For these more unquantifiable qualities some of the complicating factors and possible paths toward solution will be discussed, but no particular measurement will be given.

2.2.1.1 Repetition

The first set of qualities we will consider, those involving repetition, is probably the most difficult to quantify. Let us first examine the automatic measurement of the amount of literal repetition of distinct subparts. The most obvious obstacle here is the problem of pattern recognition. The complexity of an automated pattern search and match in a layout is greatly augmented by the fact that it is unclear at what level to look for patterns. An example will clarify this. In figure 2-23 we may consider the total figure to contain one pattern, two repetitions of a pattern, four or eight. Most observers would not have this difficulty; the ambiguity is usually

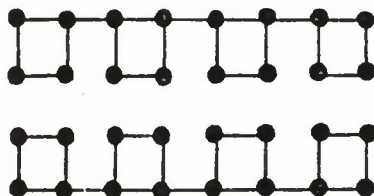


Figure 2-23

resolved by the viewer. Solving the problem of choosing patterns, then, requires some insight into the selective process by which

humans resolve this ambiguity. This is beyond the scope of this work. We can only suggest that for each viewer there is some mechanism by which he decides quite quickly that a particular pattern is in some sense maximal and minimal simultaneously and thus distinguishes it as a pattern to be matched. One factor in this decision might be that he chooses as patterns those subparts of the layout which are small enough to remember and to be easily reproduced, but large enough to be separable as a subpart of the layout.

To consider actual non-normalized measurement of the quality of literal repetition of distinct subparts, however, we must assume that the patterns of a layout have been chosen and matched, and then make our measurements on the results. The measurement is not simple even then. First we consider two obvious measurements, the number of distinct patterns and the number of instances of each pattern. As we discussed in section 2.1.1, the fewer the number of different patterns, the better the layout for comprehension. The number of instances and its effect is more difficult to judge. Here, we are concerned that the number of instances of each pattern is countable in the sense discussed in section 2.1.1. Thus we aren't interested in this number unless it exceeds some threshold and this threshold is difficult to determine.

To complete the measurement of literal repetition of distinct subparts, the patterns themselves must be examined. First we

consider their size, for, the smaller they are, the easier, in general, they will be to comprehend. But on a more subtle level, we must also consider the repetition within each pattern. In other words, if the pattern itself consists of several instances of some smaller pattern, as in figure 2-24a, it will be simpler to understand than one which does not, as in figure 2-24b. Furthermore, we must consider the relationships between patterns.

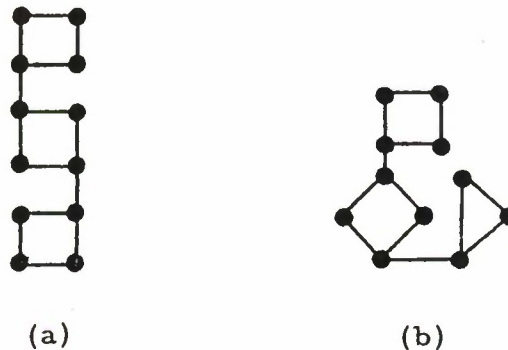


Figure 2-24

For example, are two patterns the same except for one element? Certainly, close relationships between the various patterns helps comprehension.

Thus we find that a non-normalized measurement of the amount of literal repetition of distinct subparts consists of several factors, many of which are, in themselves, somewhat unquantifiable. To combine all these factors into one measurement would be infeasible at this point.

When we consider other types of repetition, except for the very specific case of symmetrical repetition, the pattern selection and matching problem becomes worse. For the problem is complicated by the fact that if we consider non-distinct subparts we are no longer restricted to having a graph component belong to only one instance of a pattern; it may belong to several. Furthermore, when we allow for matching of pattern instances of differing sizes and orientations, the matching process becomes that much more complex. Moreover, added to the factors we must measure for literal repetition of distinct subparts, we must also consider other factors, such as size difference with similar subparts. It is clear then that finding measurements for the amounts of other types of repetition in layouts is even more difficult than for the amount of literal repetition of distinct subparts.

2.2.1.2 Symmetry

The one exception to the difficulty of measuring repetition seems to be measuring the amount of symmetrical repetition in a layout when the two repetitious subparts are to account for the whole layout. Symmetrical repetition is a very specific type of repetition (as defined in section 2.1.1). In the case that the two subparts make up the whole layout, for each possible axis of symmetry we have one pattern with two instances, one of which must be the mirror image of

the other. Therefore, a very straightforward and meaningful measurement of such symmetrical repetition in a layout, is the number of axes of symmetry in the layout. This measurement corresponds well with the extent to which such symmetrical repetition contributes to the clarity of a layout as a whole.

The one complication in automatically measuring the number of axes of symmetry in a layout is generating these axes. Since there are an infinite number of possibilities for axes of symmetry, we must use a method which somehow limits this number. Such a method has been devised for layouts with at least one link, and is given below. This method is based on the following observation. For every axis of symmetry, each line segment in a link fits one of the following descriptions:

- a) the line is on the axis,
- b) the line is perpendicularly bisected by the axis, or
- c) the line has an image under reflection with respect to the axis (this image will be called the "mirror image" of the line with respect to the axis).

Thus to find all the possible axes of symmetry, we need only consider for any one line segment in the layout:

- 1) the axis based on the line itself,
- 2) the axis which perpendicularly bisects the line, and

3) any axis with respect to which another line might be the mirror image of this line.

Each such possible axis is then checked to see whether or not, in fact, it is a real axis.

We need to describe in more detail the method used to obtain the third group of possible axes. For the chosen line segment, say $[a, b]$, we consider, in turn, every other line segment in the graph which is exactly the same length as the chosen line (since a mirror image of the line must be the same length as the line itself). For each such line of equal length, say $[c, d]$, we must examine several alternatives.

If two ends of the two line segments $[a, b]$ and $[c, d]$ coincide, then the axis which makes them mirror images must pass through this coincidental point, and the remaining two ends must be equidistant from the axis. Hence, we would obtain the dotted axis in figure 2-25a. In order for two lines without coinciding endpoints to be mirror images, there must be two pairs of endpoints (each pair having one point from each line), for example, (a, c) and (b, d) , for which the respective elements of each pair are equidistant from the axis of symmetry. Furthermore, the axis must be perpendicular to both of the lines, say, (a, c) and (b, d) , generated by these pairs (hence, the lines must be parallel), in order that line $[a, b]$ be a

reflection of $[c,d]$ with respect to the axis. Thus the pair of lines in figure 2-25b may be mirror images but the pair in figure 2-25c is excluded.

However, it may be possible that two lines may be mirror images of one another with respect to two different axes, for example, the pair in figure 2-25d. For this very special case however, it must be that the two line segments cross one another. We must remember to check both pairings of endpoints, for, although one pairing, say, (a,c) , (b,d) , might not succeed, the second, say, (a,d) , (b,c) , may, as in figure 2-25e.

The algorithm to check for these possibilities, given two line segments $[a,b]$ and $[c,d]$, then proceeds through the seven steps listed below. It must be emphasized that the success of this algorithm depends on the fact that the length of $[a,b]$ is equal to that of $[c,d]$:

- 1) Check for coinciding endpoints: if any pair of endpoints of the two line coincide, go to step 2; else, go to step 3.

- 2) Generate as a possible axis the line which bisects the angle between the two line segments. This line is determined by the coinciding endpoints and the midpoint between the remaining two endpoints. Exit.

- 3) Check the first pairing of endpoints, (a,c) , (b,d) : if the line (a,c) is parallel to the line (b,d) , go to step 4; otherwise, the axis determined by this pairing is not an axis of symmetry; therefore

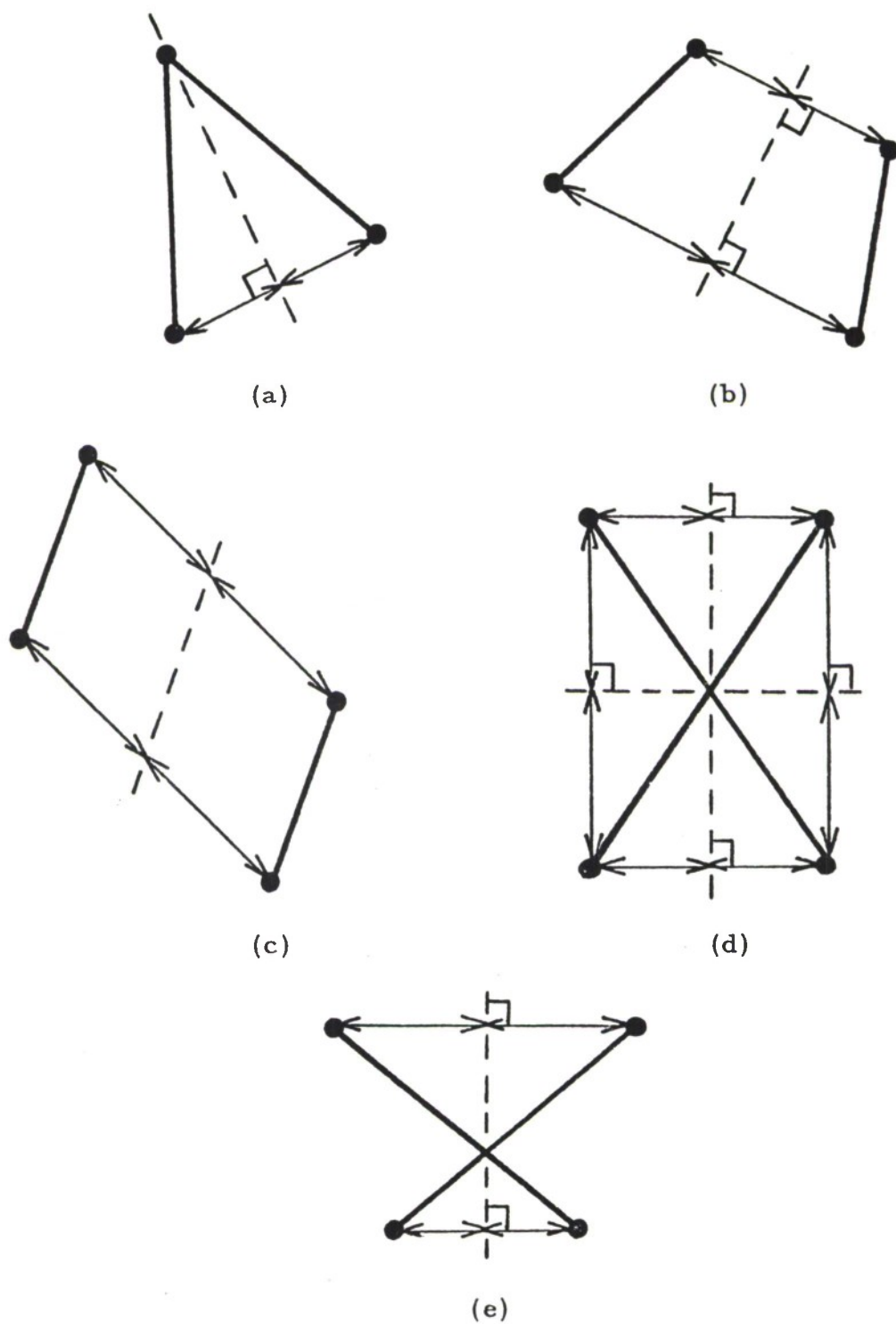


Figure 2-25

proceed to step 6, to check the other pairing.

4) Complete the check of the pairing (a,c), (b,d): if the line determined by the midpoint of (a,c) and the midpoint of (b,d) is perpendicular to (a,c), generate this line as possible axis and go to step 5; otherwise, exit.

5) If the lines (a,c) and (b,d) cross, there may be a second axis of symmetry, therefore, proceed to step 6 to check this; otherwise, exit.

6) Check the second pairing of endpoints, (a,d), (b,c): if the line (a,d) is parallel to the line (b,c), go to step 7; otherwise, the axis determined by this pairing is not an axis of symmetry; therefore, exit.

7) Complete the check of the pairing (a,d), (b,c): if the line determined by the midpoint of (a,d) and the midpoint of (b,c) is perpendicular to (a,d), generate this line as a possible axis of symmetry. In any case, exit.

The optimal way to implement this algorithm would be to take as the chosen line, [a,b], a line with the fewest equals in length in the layout. This would minimize the number of possible axes generated. Figure 2-26 depicts the generation of possible axes of symmetry for a regular hexagon using the above method, where side two is the chosen line, and where the dotted lines represent possible axes generated.

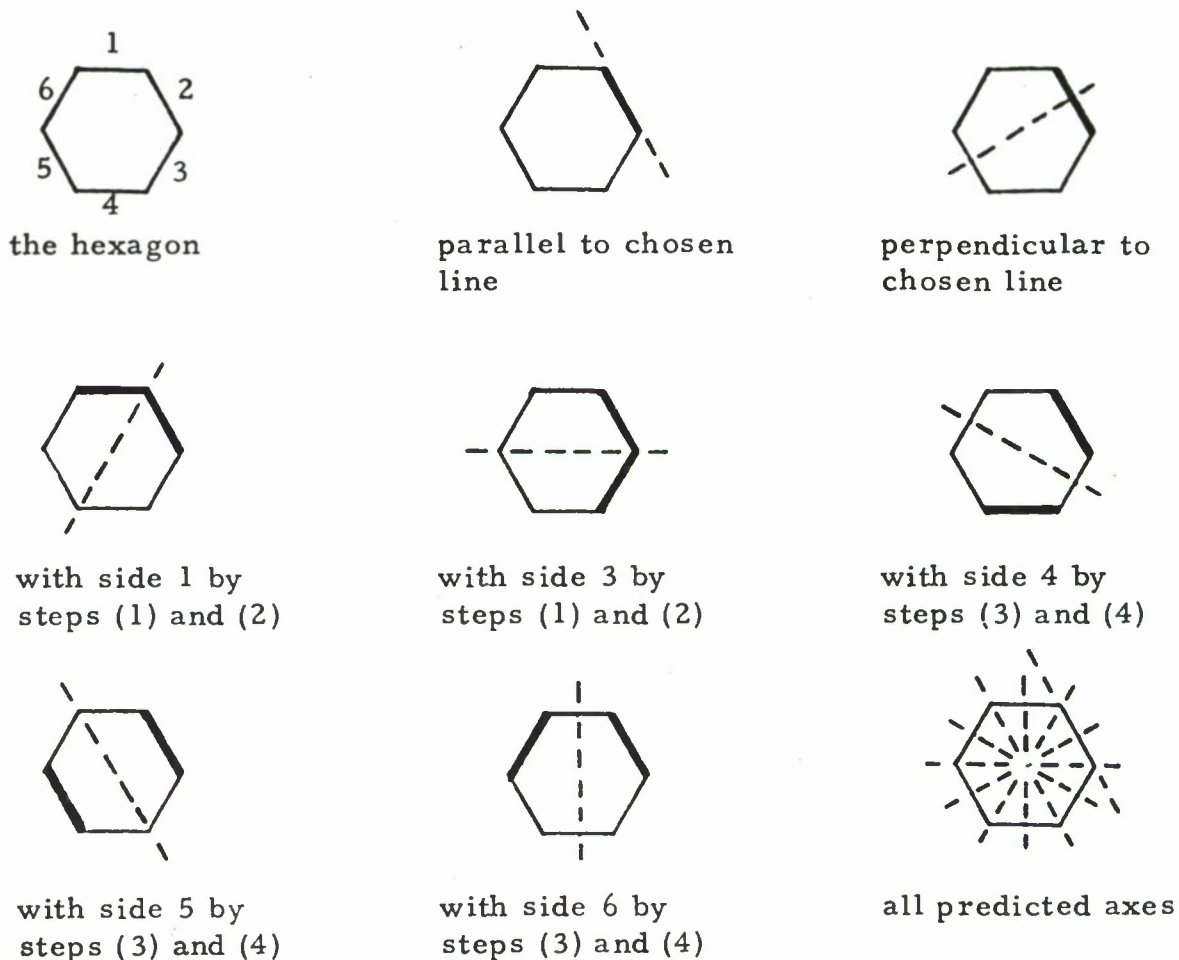


Figure 2-26

Once the axes of symmetry have thus been generated and checked, a count may be made to measure this type of symmetrical repetition in the layout.

2.2.1.3 Other Qualities of Regularity

Let us now consider measures of other qualities categorized under regularity, aside from the various types of repetition. First we will attempt to examine measurement of the effect of familiar figures. The problem here, again, is that although we may measure

certain quantities, the significance of these measurements is in question. For example, although we may measure the number of equilateral triangles that appear in a layout, this number may have nothing to do with how we perceive the layout. Compare figures 2-27a and b, for example: in 2-27a triangle recognition is key in aiding comprehension, whereas, in 2-27b, it is not. Thus we have a measurable quantity, but it does not reveal the information desired.

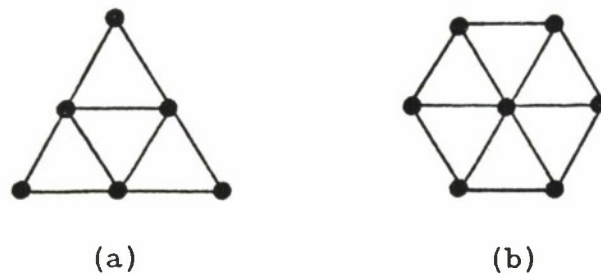


Figure 2-27

Although it detects the presence of familiar figures, it does not indicate to what extent they aid comprehension. Ideally we would like some means to pin down whether or not a familiar figure significantly determines the organization for comprehension of a layout, for example, the triangle in figure 2-27a and the hexagon in figure 2-27b. But, again, we are dealing with an extremely complex task involving the understanding of the method used by humans in resolving visual ambiguities to obtain an organization for comprehension, and this is beyond the scope of this work.

Moving on to the next quality, size and distance consistency

for single segment links, we find much more hope for significant measurement. As discussed in section 2.1.1, there are two different approaches to consistency. The first, consistency of link scale (link length consistency), might be examined by looking at the distribution of link lengths in the layout. The problem of measurement is complicated by the fact that it is not always possible to obtain total consistency in the layout for a graph. For example, it is not possible to draw all diagonals in a regular polygon the same length as the sides. The ideal measurement would first consider the minimal number of different lengths required for a particular underlying graph (for example, the complete graph on five vertices requires two lengths). It would then determine how closely the various link lengths corresponded to this ideal number. This measurement can be made clearer by an example.

Consider the layout in figure 2-28a. The underlying graph can be drawn with a minimum of two different lengths. Both figures 2-28a and b meet this requirement equally. However, figure 2-28c does not, and thus is not as consistent as possible. The layout in 2-28d, however, deviates even more from the minimum than 2-28c. Distributions of link lengths for these various layouts are shown in figure 2-29. Given that we know the minimum number of lengths for the graph, then, we might measure the deviation from this minimum in a layout by trying to answer the following questions: How close is

the number of different lengths from the minimum? If n is the minimum number of lengths for the graph, does the length distribution form n length clusters? What are the ranges in these clusters? (For example, compare figures 2-30a and b and their length distributions.)

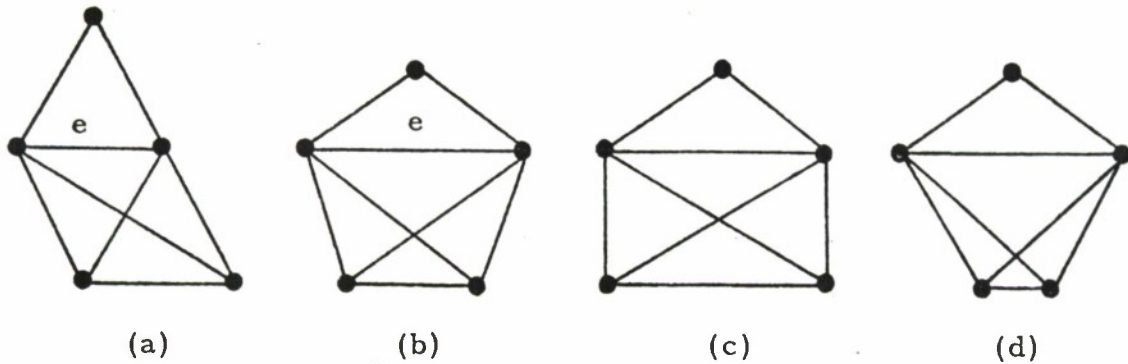


Figure 2-28

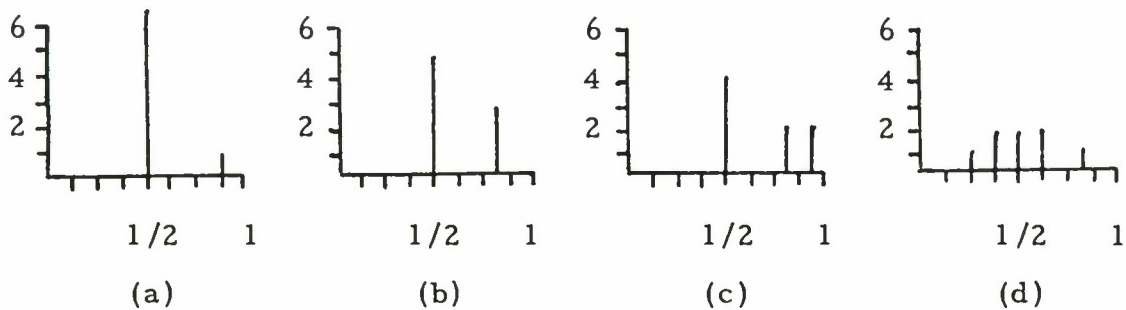


Figure 2-29

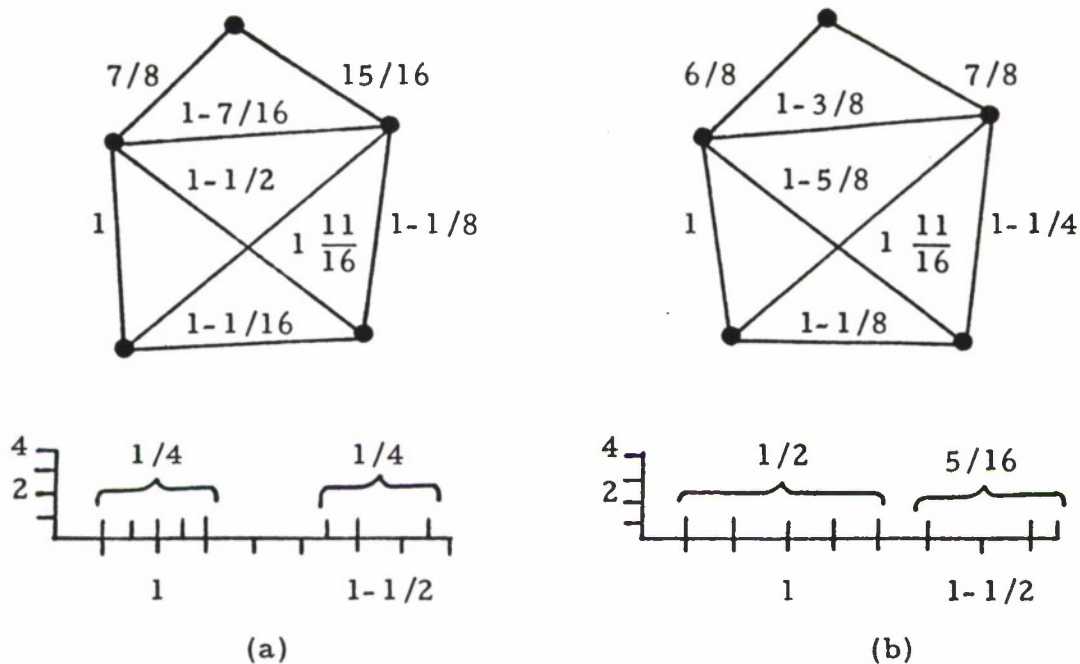


Figure 2-30

Practically speaking, however, several problems are inherent in such measurements. Foremost is the problem of obtaining the minimum number of lengths for an arbitrary graph.* There is also a question of validity in our reading of the length distribution. Suppose that length clusters overlap, or that deviations are such that there are no clusters. We cannot really get from the distributions any information which reveals which link belongs to which length

* It is clear, however, that for a complete graph on n nodes, the minimum N_n is such that:

$$N_n \leq \begin{cases} 1 + (n-2)/2 & n \text{ even} \\ 1 + (n-3)/2 & n \text{ odd} \end{cases}$$

and thus that for any graph on n nodes, the minimum is less than or equal to N_n . This result is obtained by counting the number of lengths required in a regular polygon on n nodes.

cluster. In fact, the same link in two different optimal layouts can belong to two different length clusters (for example, link e in figures 2-28a and b). Thus the problem of link scale consistency measurement turns out to be extremely complex, and should be examined further. However, for the present, we will use a simple count of different link lengths as a non-normalized measurement.

For the second approach towards examining size and distance consistency, fidelity, Baecker has designed an efficient and sensitive measurement. Several measurements were tried, and it was found that the most effective measurement was one which measured only the number of violations of the fidelity constraint, and which ignored any consideration of scale and distance deviation. The fidelity constraint is that for each link and endpoint pair, ^{*} no node with graph distance two or greater from that endpoint may be closer to the endpoint than the length of that link. The measurement then checks each link and endpoint pair in the layout and counts the number of violations of this constraint. The sum is the measurement of the infidelity of a layout.

Baecker points out that the minimum value of the infidelity measurement is not always zero. For example, he proves that the graph of figure 2-31 has a non-zero minimum fidelity. He does not concern himself with obtaining the absolute minimum for a graph,

^{*}Again, here we are dealing with single segment links.

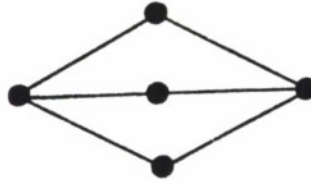


Figure 2-31

however, and finds it sufficient to observe the change in this non-normalized infidelity measurement as layouts are simplified.

2.2.1.4 Directional Consistency

The next set of measurements we consider are those involving the qualities categorized under directionality. The first quality seems the most difficult to measure, that of directional consistency in layouts. When we consider the cases in which the consistency criterion is met by the existence of a predominant direction in which arrows point (we call this the linear case), there is no problem with measurement. In flow and network diagrams, for example, to measure consistency we might measure the proportion of total link length contributing to the predominant direction. The predominant direction of the layout is determined by the direction of the vector sum of the links,* and the magnitude of the predominant direction, by the magnitude of this vector sum. The proportion of total link length

* In the following discussion of directional consistency, whenever we are dealing with links with more than one line segment, each link segment should be treated as a separate link with direction derived from that of the whole link.

contributing to this predominant direction, and thus our measurement of linear directional consistency is:

$$L = \frac{|V_s|}{\sum_i |\mu_i|}$$

where $|\mu_i|$ is the length of the i^{th} link and where $|V_s|$ is the length of the vector sum of all the links. Note that $0 < L < 1$, so that in the totally consistent case of figure 2-32a, $L = 1$, whereas with the less consistent 2-32b, $L = .8$, and with the totally non-linear case of 2-32c, $L = 0$.

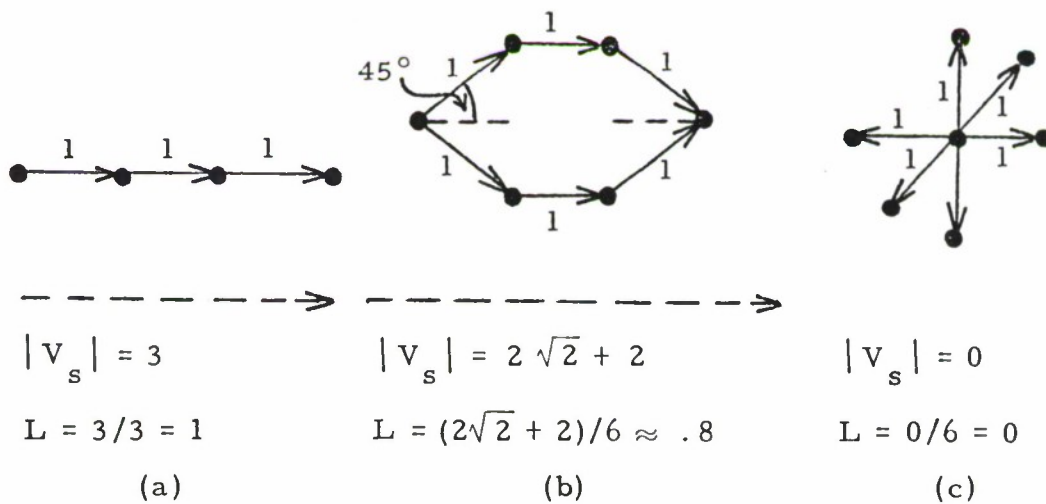


Figure 2-32

However, we would also like to find measurements for the cases in which predominant direction cannot be expressed in terms of a vector sum, the cases in which the predominant direction is radial

or circular as in figure 2-10. We note that in these cases, often vector sums tend to be near zero. However, we would like to find directional consistency measurements, analogous to that for linear consistency, for these cases. Instead of using the vector sum, then, we must find an analogous measure which sums the contributions of each of the links to radial or circular orientation with respect to a center.

Let us first consider the measurement for outward radial orientation from some fixed center. For each link, we include as the magnitude of its contribution to outward radiality its projection on the radius which runs from the fixed center through the starting node of the link. For example, in figure 2-33 links a and b have as their contribution their total lengths, since both lie on radii from the fixed center, c, and both are directed outward from c. Link d, however, has a contribution of e, since it does not lie on a radius. The length e may be determined by subtracting the distance between the end node of d and the center, c, from the distance between the start node of d and the center, c. Link f contributes the negative of its length to outward radiality, since although it lies on a radius, it is directed inward towards c. Likewise, link g contributes the negative of the quantity h; the negative value is the result of the same calculation used to find the length e. Thus this calculation also reflects link direction. In fact, if we apply this calculation to each

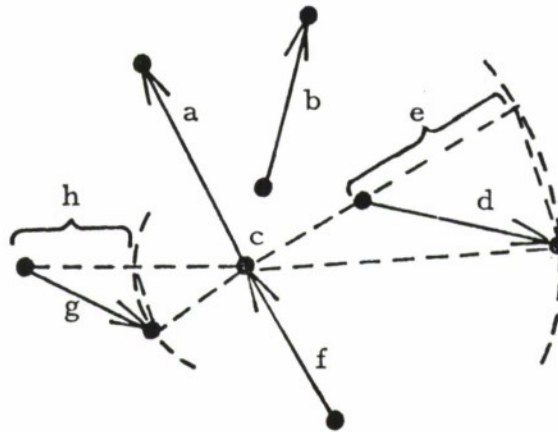


Figure 2-33

link, we obtain the outward radial contribution for that link, so that we may obtain the desired sum, σ_o , analogous to $|V_s|$ as:

$$\sigma_o = \sum_i (d(\text{center}, \text{endpoint}_i) - d(\text{center}, \text{startpoint}_i))$$

where i ranges over the set of link segments of the layout, and thus the measurement of outward radial orientation, R_o :

$$R_o = \frac{\sigma_o}{\sum_i |\mu_i|}$$

Some examples are found in figure 2-34. We note that the range for the measurement R_o is -1 to 1. This requires some explanation. In the case of measuring linearity using V_s , $|V_s|$ is always positive or zero. However, were we to examine the magnitude of the vector with direction opposite to V_s we would find it to be negative, but of the same size. When radially is measured the "vector"

opposite to the radially outward vector, $\bar{\sigma}_o$, is the radially inward vector, $\bar{\sigma}_i$. Unlike the linear case, the stronger of these two opposite directions is not necessarily selected in the process of measurement. The measurement R_o assumes that the outward direction predominates. Thus if R_o is negative, the wrong direction has been chosen as the predominant direction; we should have chosen the radially inward vector as the predominant one, and measured R_i , radially inward orientation, instead of R_o . The magnitudes of $\bar{\sigma}_i$ and $\bar{\sigma}_o$ are the same, as with V_s and its opposite, and thus we may express R_i as $-R_o$ (see figure 2-34).

One further problem should be mentioned concerning this measurement. With the linearity measurement, L was found to be very small in the case that directions diverged considerably; in other words, directions seemed to cancel each other out. There is an analogous effect with R_o (and R_i), where radially inward contributions tend to cancel out radially outward contributions. But there is also another factor which affects the magnitude of R_o (and R_i), the total amount of radiality. For example, if links were completely circular around a center, as in figure 2-34d both R_o and R_i would be zero. In the linear case the analogous effect by itself is not as marked, for the predominant direction, that of V_s , is determined by all the links, whereas, in the radial case, the predominant direction is predetermined without reference to the links. This suggests that

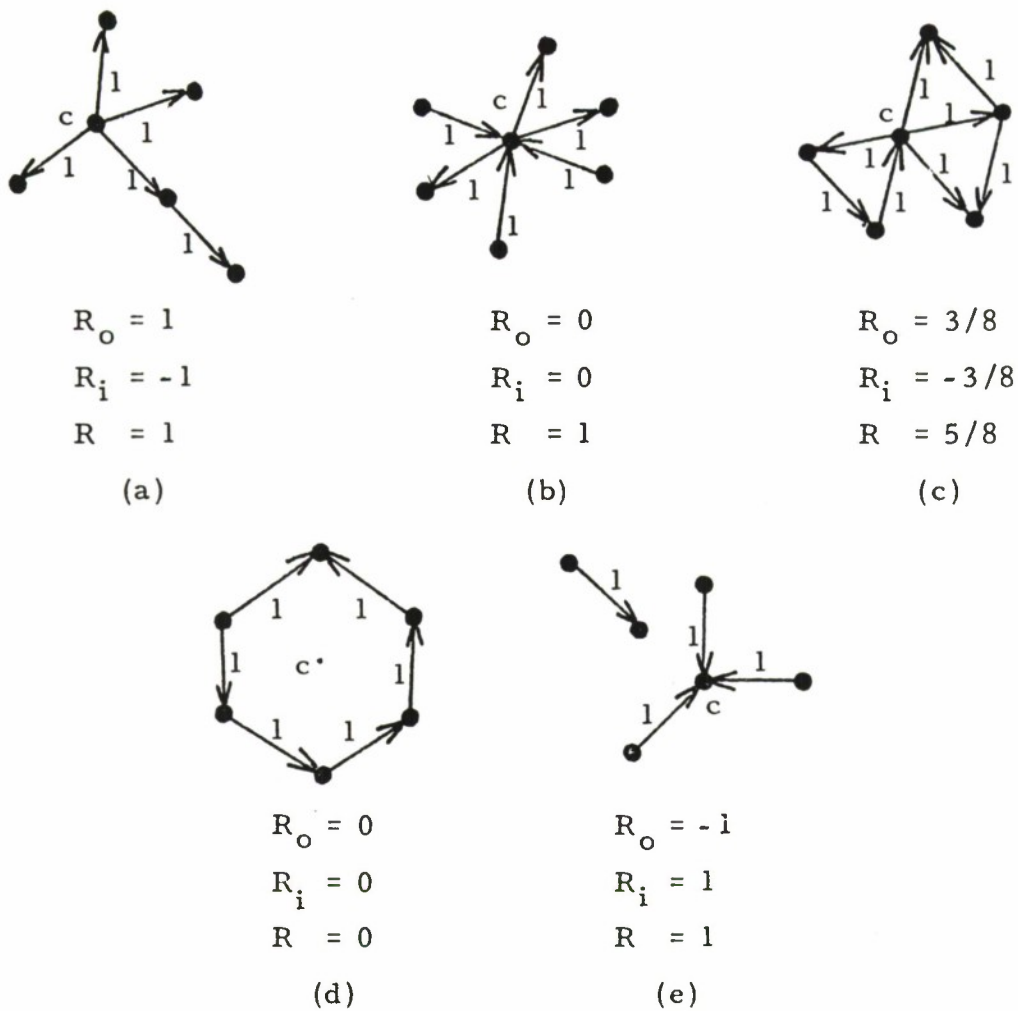


Figure 2-34

a simple measurement of radiality, regardless of direction, might be of interest also. Such a measurement is:

$$R = \frac{\sigma_r}{\sum_i |u_i|}$$

where:

$$\sigma_r = \sum_i |d(\text{center}, \text{endpoint}_i) - d(\text{center}, \text{startpoint}_i)|$$

This measurement sums contributions to radiality in either direction, inward or outward (see figure 2-34).

When dealing with analogous measurements for circularity with respect to a fixed center, c , the same phenomenon is found. First we will look at the magnitude of the "vector" sum of circular movement with respect to one direction (clockwise or counterclockwise), as determined by the sum of the contribution of each link. The contribution of a link is determined from its midpoint, m , as follows. Draw the tangent at the point m to the circle with center c , which passes through the point m . The magnitude, n , of the contribution is then the length, n , of the projection of the link onto this tangent line (see figure 2-35). Both sign and magnitude may be calculated by using $|\mu| \cos \theta$, where $|\mu|$ is the length of the link, and θ is the angle between the link vector and a vector on the tangent line oriented in the direction of the circular measurement (i.e. either clockwise or counterclockwise) (see figure 2-36).

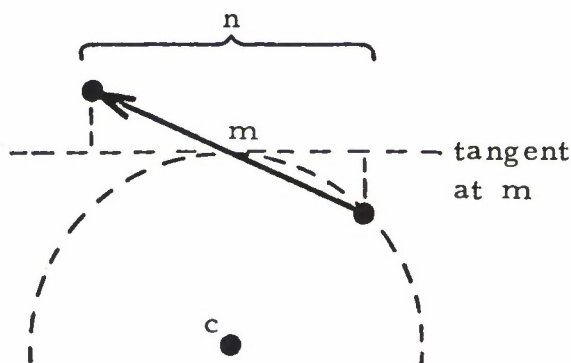


Figure 2-35

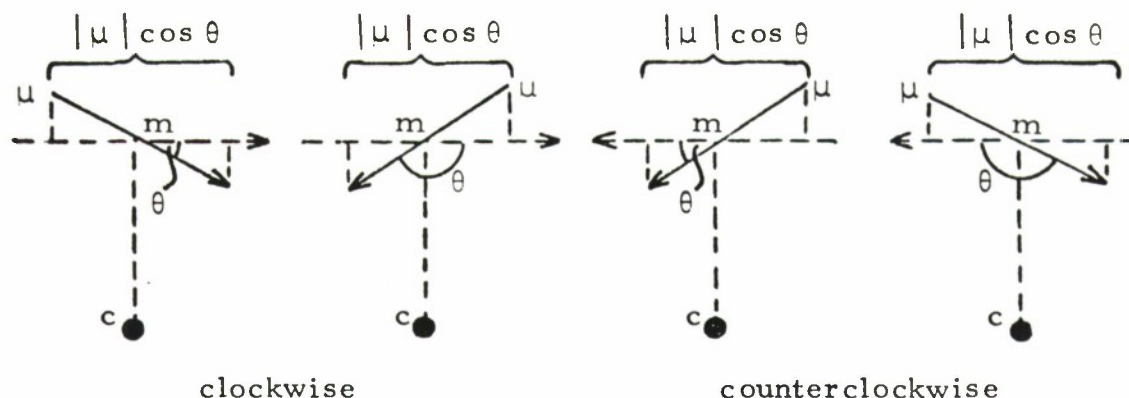


Figure 2-36

Summing these contributions, we obtain the measurement, $\sigma_{\vec{c}}$ (clockwise) or $\sigma_{\leftarrow c}$ (counterclockwise), depending on the direction of the tangent vectors used. Thus as measurements of directed circularity we have:

$$C_{\rightarrow} = \frac{\sigma_{\vec{c}}}{\sum_i |\mu_i|} \quad C_{\leftarrow} = \frac{\sigma_{\leftarrow c}}{\sum_i |\mu_i|}$$

As for R_0 and R_1 , C_{\rightarrow} and C_{\leftarrow} range between -1 and 1, and $C_{\rightarrow} = -C_{\leftarrow}$ for any layout. Analogous problems arise with C_{\rightarrow} and C_{\leftarrow} as with R_0 and R_1 resulting from the effects of cancelling and circularity. Thus we might also be interested in obtaining a measurement for undirected circularity, C , analogous to R . To do this we simply disregard sign in measuring the projections onto the tangents, to obtain:

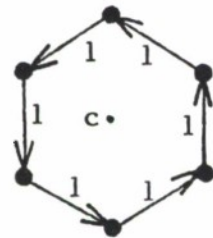
$$C = \frac{\sigma_c}{\sum_i |\mu_i|}$$

where:

$$\sigma_c = \sum_i |\mu_i| \cos \theta_i$$

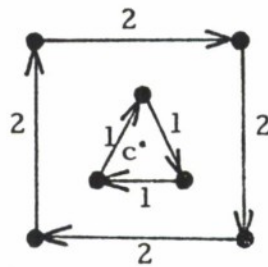
and where θ_i is the smaller angle between the link and the tangent.

Some examples are given in figure 2-37.



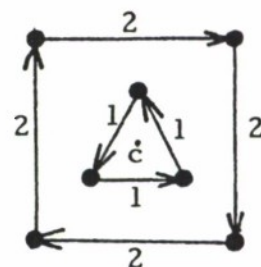
$$\begin{aligned} C &= -1 \\ C_{\rightarrow} &= 1 \\ C_{\leftarrow} &= 1 \end{aligned}$$

(a)



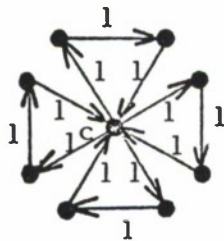
$$\begin{aligned} C &= 1 \\ C_{\rightarrow} &= -1 \\ C_{\leftarrow} &= 1 \end{aligned}$$

(b)



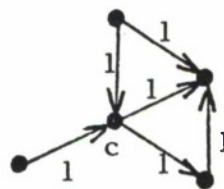
$$\begin{aligned} C &= 5/11 \\ C_{\rightarrow} &= -5/11 \\ C_{\leftarrow} &= 1 \end{aligned}$$

(c)



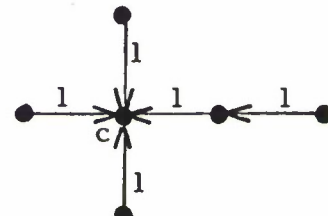
$$\begin{aligned} C &= 1/3 \\ C_{\rightarrow} &= -1/3 \\ C_{\leftarrow} &= 1/3 \end{aligned}$$

(d)



$$\begin{aligned} C &= 0 \\ C_{\rightarrow} &= 0 \\ C_{\leftarrow} &= 1/3 \end{aligned}$$

(e)



$$\begin{aligned} C &= 0 \\ C_{\rightarrow} &= 0 \\ C_{\leftarrow} &= 0 \end{aligned}$$

(f)

Figure 2-37

2.2.1.5 Other Qualities of Directionality

Measurement of the remainder of the qualities categorized under directionality is quite straightforward. For example, as non-normalized measurements for the number of bends, total link length, and the number of intersections, we may simply use these numbers. It is interesting to note that Fary (14) has proven that for planar graphs, with no self-loops or parallel links (two links connecting the same two nodes), there always exists a layout which has no bends, or equivalently, one in which all links are represented by single line segments.

The question of a normalized measurement for the number of intersections involves a well-known problem in graph theory, that of determining the genus of a graph. This problem has been examined by several people including Anger (2). A modified version of the problem, that of determining whether or not a given graph is planar, has also been of interest in graph theory, and has been examined by Even et al. (13) among others. Anger's method to find the genus of a graph constructs layouts with all possible permutations of link orders around the nodes. This produces all possible representations of a graph with respect to genus. The genus of each such layout is examined, and the layout with the smallest genus determines that of the graph. The method may be quite time consuming, and Anger mentions several ways in which it may be made more efficient. Thus,

if we wished to find the value of the normalized measurement of intersections in a layout, it would first be necessary to determine the genus of the underlying graph using a method such as Anger's. To answer the question of whether or not the genus is zero, we need only use an algorithm such as Even's. Even's algorithm is more direct and less time consuming than one which determines genus. And it is constructive in that the results of the check can be used to generate a planar layout, if the graph is planar.

There is an additional problem in measuring the number of intersections. As mentioned in section 2.1.2, it may be the case that a smaller number of intersections is not necessarily best. Considering this, we would ideally like to measure not the number of intersections, but how much the intersections present add or detract from directionality in the layout. Such a measurement is very difficult to make, however. One indirect approach might be to make the judgment based on the amount of complication which develops when the intersection or set of intersections is removed. We might examine the increase in link length and number of bends in this case. For example, when we eliminated the intersections in figure 2-15b, the result, 2-15c, had four more bends and some increase in link length. Here, in order to measure how much the intersection adds or detracts from directionality, we are measuring how much other complication we avoid by allowing the intersection, keeping everything else

constant. This seems a feasible approach to the measurement in a local sense. However, when we consider such a measurement without keeping other layout factors constant, such as node position, the measurement would become extremely difficult to devise.

Several measurements may be developed to quantify the amount of parallelism or the number of different link segment slopes in a layout. We may, on the simplest level, consider counting the number of different slopes. Or, we may also take the number of link segments into account in the measurement, and use a measurement such as:

$$N = \frac{\text{number of slopes } (= N_s)}{\text{number of link segments } (= N_l)}$$

Here N ranges between 1, in which case every link segment has a different slope, and $1/N_l$, in which case all link segments are parallel. We will not consider measuring parallelism separately, since, in a sense it is reflected adequately in the measurement N . When N_s is small with respect to a given N_l , this implies that the parallelism is large.

An interesting question to ask is how the number of link slopes relates to the phenomenon of directional consistency, and furthermore, should we consider a measurement of the number of slopes similar to the measurements used for directional consistency. In measuring

directional consistency we are trying to determine how well the various links combine to give a total direction to the layout. In examining the number of slopes, on the other hand, we wish to determine how much divergence (in a discrete sense) there is between link directions; it is not a measurement of total direction, but of relationships between the link segments. Thus, in measuring the latter, we do not want to take total direction into account, but only whether or not link directions agree.

Finally, we would like to obtain a measurement of horizontal-vertical link segment orientation in a layout. There are two senses in which we may measure this, corresponding to the two interpretations of link slope just discussed. In the first case, using a vector type analysis, we would be asking what contribution each link gives to both horizontal and vertical movement in either direction. Since these are perpendicular directions, this sum accounts for all orientation, and the answer is meaningless. In the second case, we would simply examine the ratio of the number of strictly horizontal and vertical segments, N_{hv} , to the total number of link segments, N_s .^{*} This approach is preferable, not only because it gives a meaningful result, but also because, again, we are interested in a discrete relationship,

^{*} Or, we might measure the ratio of total horizontal and vertical length to total length. This gives the percent of total link length which is horizontally or vertically oriented.

that between the links and the horizontal and vertical axes. In other words, in this measurement, we are not concerned with how close a link is from being horizontal or vertical, but whether or not it actually has one of these orientations.

2.2.2 Realization

In dealing with the problem of realization in graph layouts of the qualities discussed above, there are two main approaches which might be taken. The first, which we shall call the constructive approach, includes realization techniques which aim at providing layouts with the absolute optimum for the qualities under consideration. With this approach a layout is developed from the graph, without reference to any previous layout for the graph, but based only on the criterion that a quality or set of qualities chosen be optimized.

The second approach, called the modifying approach,^{*} aims at realizing the local optimum for the qualities being considered, given an already existing layout for the graph. Methods developed with this approach apply to a given layout in order to improve that layout as much as possible with respect to a given quality, while changing the layout as little as possible. This constraint that as

^{*} This terminology should not be confused with the ideas presented in chapter 4. A "modifying algorithm" applies to a layout in a global sense, whereas the "modified layout problem" (as presented in chapter 4) deals with layouts in a very local sense.

little as possible be changed with the modifying approach is based on the fact that with these algorithms we want to preserve as much of the original layout as we can while optimizing for a quality.

Although the constructive approach is initially more appealing than the modifying approach, since the results promise to be better, there are many problems which accompany it. In order to provide the greatest amount of flexibility of layout, the constructive approach requires that algorithms use no reference to any previous layout. The problem here is that given a graph and a set of layout qualities to be optimized there may be several layouts for this graph which optimize this set of qualities. In other words, with the constructive approach, part of the resulting layout may be arbitrary. For example, in figure 2-38, we see several layouts for one graph which meet the requirement of minimal number of different link lengths (link length consistency). Thus this link length consistency requirement is not enough to determine, for the graph underlying figure 2-38, which layout a constructive algorithm should produce, and an arbitrary decision must be made. With the modifying approach, no arbitrary layout decisions are made; as mentioned above the implication behind this approach is that, given a layout, as little as possible in that layout should be changed in optimizing for a quality.

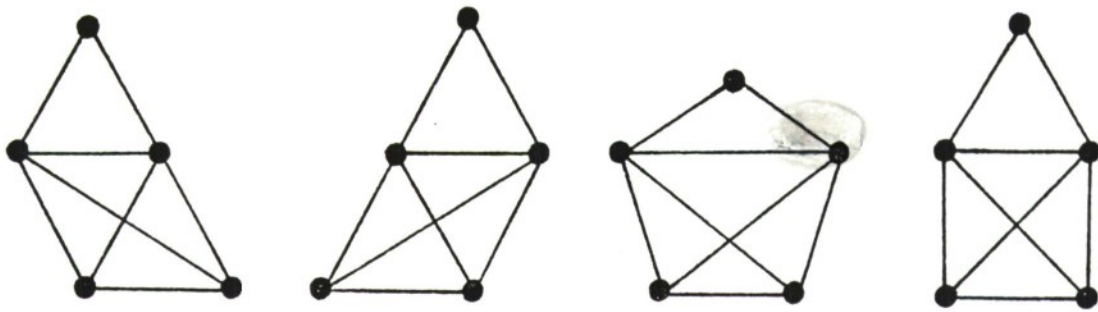


Figure 2-38

A second problem to consider with the constructive approach is the problem of combination of criteria. We can apply a constructive realization algorithm only once to a graph to obtain a layout, since it must always be applied to a graph, and not to another layout. A second application obliterates the results of a first application. And if we wish to optimize for more than one criterion, we must combine the respective constructive realization algorithms into one. The difficulties then begin to multiply, for, first of all, this means that we must obtain a different algorithm for each combination of criteria, and second, it may be the case that for a particular graph, two criteria may conflict. For example, in figure 2-12a, we optimize the layout for minimal number of intersections, and in 2-12b for minimal link length and number of bends. We cannot, for this underlying graph, have all three measurements be at absolute minima simultaneously.* This implies that we must assign priorities to the

*This problem of interaction and trade-offs between qualities, where they conflict in layout, is an important problem in itself, and should be looked into further.

criteria in a constructive realization algorithm. The problem here, however, is that we often cannot know, before experimenting with several layouts, what criteria should be given priority and how one criterion affects another. Furthermore, the addition of choices of priorities would increase the complexity of these algorithms a great deal.

With the modifying approach, we do not have these problems, since these algorithms apply to already existing layouts. We need only consider one algorithm for each quality, and apply these algorithms to the layout one at a time. This puts the concept of algorithm definition at a much more feasible level. There is also an advantage in that we may examine the results of optimizing for a particular quality more closely, and observe how it affects other qualities in the layout.

This modifying approach also provides the possibility of a somewhat interactive environment in which, perhaps, a user may try several different orderings and combinations of qualities for optimization, to find the ones which are most appropriate for his particular graph. The modifying approach also seems much more reasonable in that it provides a good environment for testing more than one method to optimize for a given quality. This ability to experiment which is facilitated by such an environment is a very important consideration at this point in our research. For example, we might

wish to experiment with the extent to which we allow an algorithm to change a layout for optimization purposes.

To complete the comparison of these two approaches, we must also take into account the difficulty mentioned in section 2.2.1 in measuring the absolute optimum for many of the qualities considered given a graph. Since a realization of such an optimum is the inherent goal of constructive realization algorithms, they must be able to find this optimum. With the modifying approach this problem is avoided. Essentially we ask that these algorithms do the best they can in optimizing, while limiting the power of the particular algorithm to change a layout. Thus, we are not concerned with what the absolute optimum may be, but what the local optimum is, given certain constraints.

For these reasons the modifying approach is the one taken in the discussion of realization algorithms which follows, as well as in the experimental MOD system described in the next section. For a few qualities, however, there are some constructive algorithms which are of interest. These will be mentioned.

There are many difficulties in the development of realization algorithms for the various qualities discussed in section 2.1, as was seen in the discussion of measurements for these qualities. Again, where we cannot actually provide an algorithm for realization of a specific quality, we will attempt to point out some of the problems involved.

2.2.2.1 Repetition

As in the case of measurement, the first quality we will consider, repetition, is the most difficult to deal with. Here the aim is to find a method for changing a layout so that more repetition occurs in the layout. The complication in dealing with repetition, as discussed in section 2.2.1.1, is the discovery of matching patterns in the layout and the decision as to the level at which to select patterns. With realization, however, the problem is even worse than it is with measurement since we are not looking for pattern matches which presently exist in the layouts, but for possibilities of creating them. Were we to try to develop a modifying algorithm, this would perhaps mean identifying "near" matches and making them "perfect" matches. Finding a constructive algorithm would involve identifying structural matches in the graph and realizing them in layouts. Both tasks seem infeasible at this point.

However, since we have chosen to work in a somewhat interactive environment, a compromise is possible. The most difficult part of the task to automate is the identification of possible matches. The most tedious part for the user is the realization of the repetition. Thus, if we break the task up and allow the user to specify the subparts of the layout to be matched, and allow a modifying algorithm to actually make the subparts repetitious, we have a reasonable compromise. With such an approach to the realization of repetition there

are several things to consider. Foremost is the question of node-to-node mapping from one subpart onto another. An example can best explain this. Suppose in the layout of figure 2-39a that the user specified, with appropriate enclosures, the two subparts to be matched (these need not be distinct). This is not a complete specification since he has not explicitly identified nodes. It seems quite clear, however, that he means that node *a* is to be mapped onto *a'* etc., if he wishes to make the two patterns repetitions of one another as in figure 2-39b. But suppose he wishes to make them symmetric images of one another with respect to an axis between them. Then the mapping would be different, as indicated in 2-39c. Thus the user should specify in which sense (literal or symmetrical) he wishes to have the repetition realized, but he may leave to the realization algorithm the task of making the node-to-node map given this

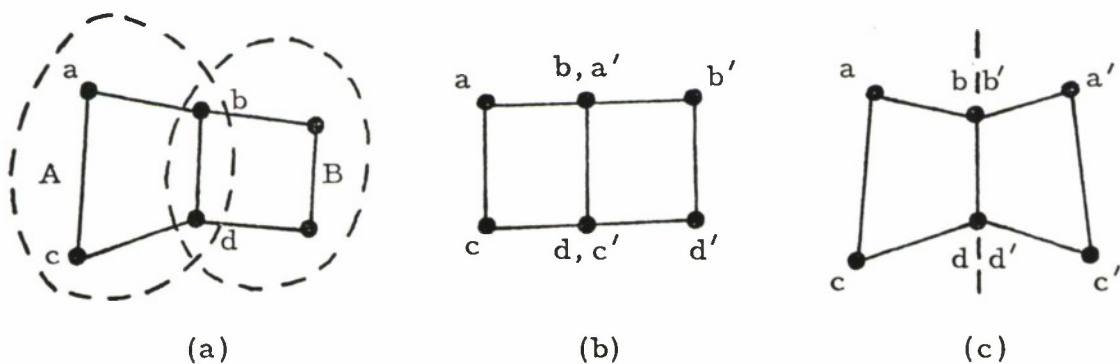


Figure 2-39

information. In fact, with either literal or symmetrical repetition, there may be more than one mapping which will suffice. This will be brought out later in the discussion.

Another question of importance is whether, in changing the layout to produce repetitious figures, we should use one of the specified subparts as a pattern and require that the other conform to it, or, whether we should find some compromise between the two to which we make both subparts conform. If we allowed only the first alternative, and chose subpart A in figure 2-39a as the fixed pattern, we could not make subpart B a literal repetition of it, as can be seen in the figure. If we followed the other alternative, we would have a problem when we wanted to make several subparts the same by using this operation which applies to only two subparts at a time. Thus, the success of the manner in which we realize repetition depends on what the user desires. It should thus be left to him to choose between the two methods according to what he has in mind.

In summary, then, we have sketched the following requirements for our repetition algorithm. Given, by the user:

- (a) two layout subparts (designated in some way, for example, by enclosure),
- (b) whether we want to make the subparts literal or symmetrical repetitions of one another, and

(c) whether the first subpart designated may be modified, perform the following:

(1) Find isomorphic node-to-node maps between the subparts, which are appropriate to the choice in option (b). Establish the axis of symmetry for each mapping, when symmetrical repetition is requested.

(2) Dependent on the choice in option (c), attempt to make the two subparts either literal or symmetrical repetitions of one another by either:

(2a) changing the second specified subpart to conform to the first, or

(2b) finding an intermediate form between the two subparts and changing both to this new form.

In some cases, it is obvious that literal or symmetrical repetition is not obtainable; in these cases the algorithm should terminate without making any changes. For example, if we were to require that subparts A and B of figure 2-39a be made literal repetitions of one another without changing subpart A, the algorithm should fail. In other cases, it may be that the only solution is one in which the two subparts overlap; we also want the algorithm to fail in these cases.

Such an algorithm has been written, and, due to its length, has been placed in Appendix 2, rather than in the text. The appendix

includes the algorithm, along with a detailed explanation and some examples of its application.

Several problems remain with the algorithm as stated in the appendix, and should be looked into further. The foremost problem is prevention of overlap of two nodes in the resultant layout. The algorithm does not necessarily prevent this. In fact, it is possible that in moving nodes to produce literal or symmetrical repetition, with certain layouts, two nodes might be placed at the same location. The algorithm in its present form, contains no checks for this, although, a more detailed version might include such checks.

Another problem is that we have included no provision for similar and rotational repetition. It seems in cases like that of figure 2-39a, were we to require that B be made a literal repetition of A without moving A, the result, since literal repetition is not possible, should be similar repetition, if this were possible. Similar repetition has a natural place in such an algorithm as a default condition when literal repetition cannot be accomplished. Again a later version of the algorithm might include a provision for similar repetition in this form. Despite these problems, the algorithm in its present form, reflects a feasible approach to realizing repetition in layouts.

2.2.2.2 Familiar Figures

The next problem to consider is the realization of familiar figures in layouts. Again the problem is that it is difficult to recognize automatically when a familiar figure, such as an equilateral triangle, is possible and appropriate in a layout. Once this has been established, it is quite easy to realize the figure. Thus, an approach similar to that used for repetition might be appropriate, where the user indicates the subpart of the layout he would like to have appear as a regular figure, and an algorithm performs the mechanics to produce it. For example, the user might indicate some simple cycle of the layout, and the algorithm would move the nodes of the cycle minimally to obtain a regular polygon. The one exception to the problem of recognition of familiar figure possibilities seems to be recognition of horizontal and vertical lines of nodes in a figure. It is quite simple to detect when a series of nodes in a layout is intended to fall in a straight horizontal or vertical line, and to modify the layout so that the nodes are aligned in this way. Such an algorithm has been implemented in the MOD system and is described in section 2.3.3 under the pretty command.

2.2.2.3 Link Length Consistency

We would now like to find a method to optimize link length consistency in layouts. Using the number of different lengths in a layout as a measurement of length consistency, one such procedure might be

as follows. For each nodes (and bend point) in turn find a new position for the node (leaving other nodes fixed), which minimizes the number of lengths. Continue until one complete pass through the nodes yields no improvement in the measurement.

The heart of the algorithm is the determination of a position for the node under consideration which minimizes the number of link lengths. The constraint is implied in this step that we move the node as little as possible, in order that the layout is disrupted minimally. Suppose the node under consideration, say a , is of degree n (i.e., has n links attached to it). And suppose that of the links not attached to a in the layout, there are m different lengths, ℓ_1, \dots, ℓ_m . We will try to find a position for the node a , such that each of the n links is equal to one of the m lengths. There may be several such positions. To find these positions, we first try assigning the n links so that they all will be equal to the same one of the m lengths, and then derive a position for the node a which satisfies this assignment. If no such position can be found, we try assignments which use only two of the m lengths; if this fails, try three, and so on. Once we find a successful assignment and position using p of the m lengths, the process is terminated for the node under consideration.

An assignment is considered to fail if the set of equations described below has no solution. Let x, y be the new position to be derived for node a . Let x_i, y_i ($i = 1, 2, \dots, n$) be the position of the

node at the other end of the i^{th} link adjacent to a , and let ℓ_i ($i = 1, \dots, n$) be the length (one of m) assigned to link i . Then we must find a value for x and y which satisfies each equation in the set:

$$\{ \sqrt{(x_i - x)^2 + (y_i - y)^2} = \ell_i; \quad (i = 1, 2, \dots, n) \}$$

In the case that $n = 1$, the new position, x, y of node a will be on the line $((x', y'), (x_1, y_1))$, where x', y' is the old position of node a .

Where the position x, y is not uniquely determined by this set of equations, the position closest to x', y' is taken as the solution. Thus, an assignment fails if no position can be found for node a which allows links $1, \dots, n$ to have lengths ℓ_1, \dots, ℓ_n , respectively.

It is possible that if we are using p of the m lengths there might be several assignments which can be satisfied. We make a selection among these by choosing that assignment which yields the new x, y closest to the old position x', y' . The number of different assignments of m lengths to the n links where we want to use p of the m lengths is given by $N_{p, n, m}$, as follows:

$$N_{1, n, m} = m$$

$$N_{p, n, m} = \binom{m}{p} p^n - \sum_{i=1}^{p-1} \binom{p}{i} M_{i, n, m}$$

where:

$$M_{p,n,m} = N_{p,n,m} / \binom{m}{p}$$

An example will help to explain the procedure better. Suppose node a with position x', y' has three links adjacent to it, and that there are three lengths in the remainder of the layout, say ℓ_1, ℓ_2 , and ℓ_3 . We first try the assignments for which $p=1$. There will be $N_{1,3,3} = 3$ of these:

Link	Assignment		
	(i)	(ii)	(iii)
1	ℓ_1	ℓ_2	ℓ_3
2	ℓ_1	ℓ_2	ℓ_3
3	ℓ_1	ℓ_2	ℓ_3

For each of the three assignments ($i=1, 2, 3$) we must solve the equations:

$$\{\sqrt{(x_k - x)^2 + (y_k - y)^2} = \ell_i; (k=1, 2, 3)\}$$

Suppose, then, we obtained solutions for all of these, say respectively (x^1, y^1) , (x^2, y^2) , and (x^3, y^3) . We would then choose the assignment i for which:

$$\sqrt{(x' - x^i)^2 + (y' - y^i)^2}$$

was minimal, and go on to the next node.

If, on the other hand, no solution was found to any of the equations for $p=1$, we would have to try the $N_{2,3,3} = 18$ assignments for

which $p = 2$, proceeding in the same way we did for $p = 1$, to find the solution which changes the position of node a the least. If we proceed through $p = \text{Min}(m, n)$ without finding any solutions, we leave the node at its original position and move on to the next node.

For each node considered, if at least one of the lengths of its adjacent links was unequal to any one of the m lengths of the remaining links, and if a successful assignment has been found for this node, then we have improved the measurement of consistency, since the total number of link lengths will be decreased to m . In any other case, in other words, in the case that all the adjacent links originally had one of the m lengths, or, in the case that no successful assignment was found for the node, the measure of link length consistency remains the same.

This algorithm has not yet been implemented or experimented with. Thus, its performance and its pitfalls are not known. For example, perhaps we should also allow assignment of only some of the n links for a node, leaving some fixed as they are. Or perhaps, when the n links are already of equal length, we should not change them. One obvious problem is that for large n or m the number of trials for a given p becomes very large. Figure 2-40 depicts an example of the application of the algorithm as it stands. Figure 2-41 shows a table of the steps performed for this layout. The checked column indicates the one assignment chosen from those of the

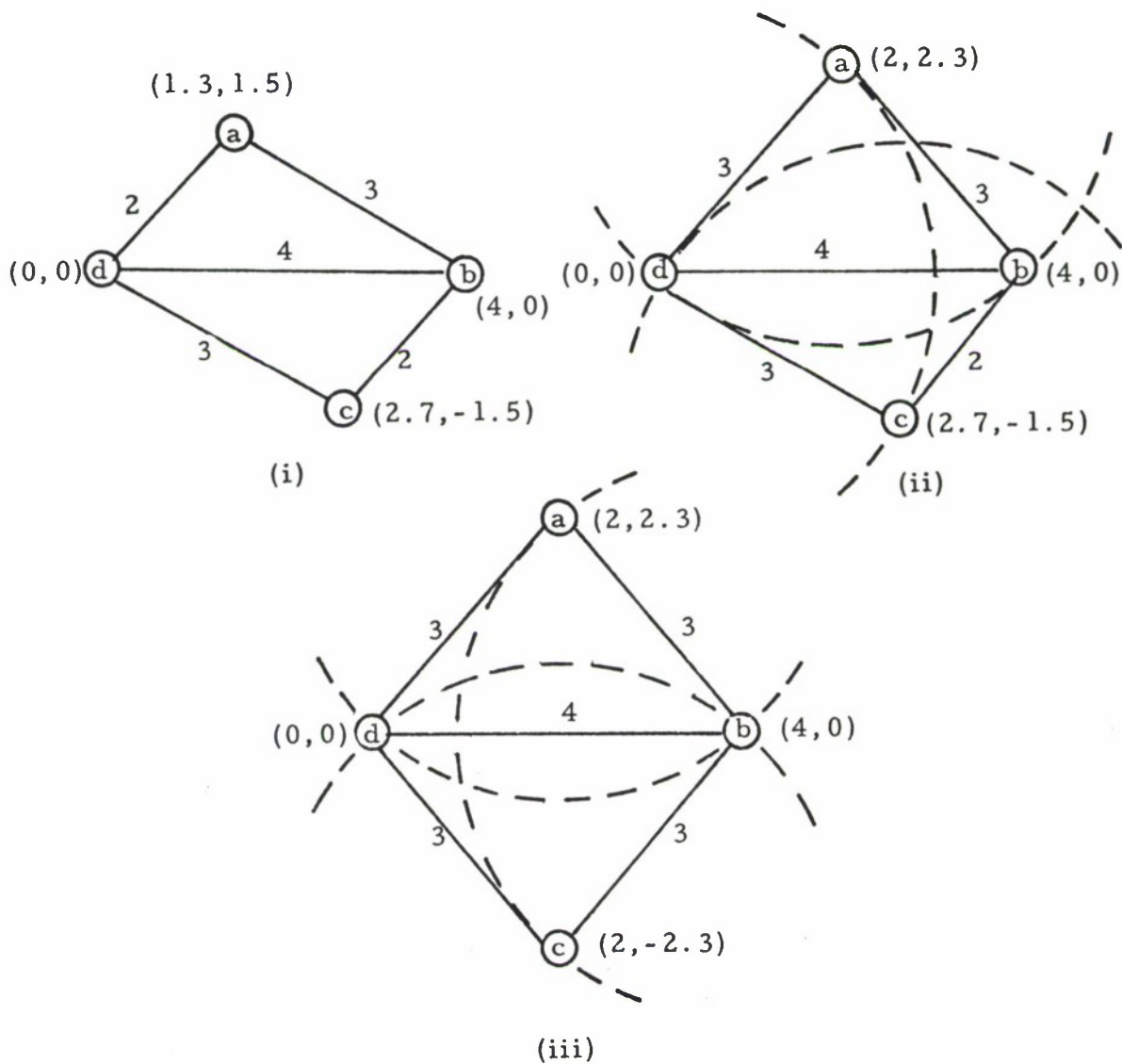


Figure 2-40

$N_{p,n,m}$ which are successful.

An alternative approach to the problem of realizing link length consistency has been examined, and an algorithm has been sketched out in Appendix 3. This approach is more of a constructive algorithm (as defined in section 2.2.2) than the one given above, although it

<u>Step Node</u>	<u>n</u> <u>m</u> <u>p</u>	<u>Assignments</u>	<u>Resultant Position</u>	<u>Consistency</u>
(1) a = (1.3, 1.5) figure 2-44 i	2 3 1	a-b 4 .3 2 a-d 4 3 2 x 2 2 2 y 3.5 2.3 0 ✓	x = 2 y = 2.3	3
(2) b = (4, 0) figure 2-44 ii	2 1 1	b-a 3 b-d 3 b-c 3 x } y } no solution	x = 4 y = 0	3
(e) c = (2.7, -1.5) figure 2-44 ii	2 2 1	c-b 4 3 c-d 4 3 x 2 2 y 3.5 -2.3 ✓	x = 2 y = -2.3	3
(4) d = (0, 0) figure 2-44 iii	3 1 1	d-a 3 d-b 3 d-c 3 x } y } no solution	x = 0 y = 0	2
(5) a = (2, 2.3) figure 2-44 iii	2 2 1	a-b 4 3 a-d 4 3 x 2 2 y 3.5 2.3 ✓	x = 2 y = 2.3	2
(6) b = (4, 0) figure 2-44 iii	3 1 1	b-a 3 b-d 3 b-c 3 x } y } no solution	x = 4 y = 0	2
(7) c = (2, -2.3) figure 2-44 iii	2 2 1	c-b 4 3 c-d 4 3 x 2 2 y -3.5 -2.3 ✓	x = 2 y = -2.3	2

Figure 2-41

uses some of the original layout, and so should be classified as a modifying algorithm. The idea behind it is to break the layout up into cycles of minimal length, and to generate regular polygons for these cycles wherever possible. It is guaranteed that the resultant layout for n nodes will contain not more than N_n^* lengths. The algorithm sketched is quite complex and several problems still remain in its design.

2.2.2.4 Fidelity

In his paper Baecker has sketched a feasible approach to realization of higher fidelity in a layout. In summary, given a layout, he suggests four heuristics, the application of which should improve fidelity:

1) Isolate "maximal dangling trees," those non-cyclic components obtained by separating the graph only at articulation points^{**} which lie on simple cycles.

2) Move nodes of large degree in the direction of the vector sum of the links from this node, taken in the direction away from this node.

*
 Again,
$$N_n \leq \begin{cases} 1 + (n-2)/2 & n \text{ even} \\ 1 + (n-3)/2 & n \text{ odd} \end{cases}$$

where n is the number of nodes and bend points.

**
 A node in a graph is called an articulation point if, by deleting the links adjacent to this node, the remainder of the graph is separated into two or more components.

3) Reduce intersections by local node transversal of links; for example, if all the links from one node cross some given link, move the node to the other side of the given link.

4) Identify interesting subpatterns and manipulate them. This is the least specific of the heuristics, and implies manipulations such as recognizing poorly placed whole subgraphs and repositioning them.

Baecker has derived these heuristics from observing the manual manipulation of layout, and finds them quite successful in improving fidelity. He does not, however, attempt to put them in the form of an algorithm, although this might easily be done for the first three heuristics.

2.2.2.5 Directional Consistency

We will now move on to a discussion of the realization of directional consistency for directed layouts. First we consider realization of linear directional consistency as discussed in section 2.2.1.4. A direct approach to the improvement of the measurement of linear directional consistency, L , might be accomplished by continuously alternating the following two operations; first, determine the most deviant link in the current layout, and, second, rotate this link so that its direction is less deviant, thus obtaining a new "current" layout. A repetition of the following three steps would accomplish this:

1) Find the direction of the vector sum in the layout; if $|V_s| = 0$ choose the direction of any one of the links.

2) Find the link not yet tried for this iteration for which the product of the link length times the angle at which it deviates from the direction chosen in step 1, is the largest, and call this link i . If all links have been tried, the realization process is terminated.

3) Using the center point of link i as a pivotal point, and keeping its length constant, rotate the link and its two endpoints until it lies in the same direction as the direction chosen in step 1. If this new position causes nodes to overlap over other nodes or links, or, if the resultant L is not less than the previous L , return the link to its previous position and go back to step 2; otherwise the new position is kept, and the iteration is complete.

With each iteration through these three steps either we obtain an improvement in the measurement L , or the iteration process terminates. We note that the overlap condition must always be checked since an optimal solution in some cases might be for some links to lie on top of others. For example, with the layout in figure 2-42i an L of one would be obtained if the links on the upper part of the layout and the links on the lower part of the layout were all placed along the line through the center of the layout, in which case there would be overlap.

There are several problems inherent in this method which must be explored. First is the problem of determining when a sufficient number of iterations have been performed. Perhaps, if we measure L after each iteration, we might terminate the iteration process when improvement begins to grow small, although we have no guarantee that the improvement is a monotonically decreasing function. Furthermore, we do not know how to define "small."

Second is the question of whether or not the optimal new position for a link i , chosen in step 2 of an iteration, is in the direction chosen in step 1. Perhaps some intermediate position between the old and the new direction might be more optimal and should be used. In an actual implementation of the algorithm, this possibility should definitely be explored.

A third question is whether, in fact, the most deviant link should be the one adjusted first in an iteration. Certainly, reorienting this link in the direction of V_s adds to the linear directional consistency, L . However, other links, namely those attached to the endpoints of the link i are also affected, and their modification may detract from L . Considering this, perhaps, in step 3 we should, instead, choose first the link whose movement would cause the greatest net improvement in L . Thus we would have to perform step 3 for each link in order to make our decision in step 2, choosing to move that link with the greatest net increase in L for this iteration. At

this point this approach seems quite time consuming. The relative performance both in time and in quality of results of this modified method, compared to the one originally described, might be judged best by a comparison of their performance upon implementation.

One further possibility should be mentioned, that of allowing an iteration to decrease L , in anticipation of a later net increase. In other words, by restricting ourselves only to modifications which increase L , we may be restricting the possibility of improvement to a very localized range. Whereas, perhaps, were we to allow L to be decreased in an iteration (and backtrack upon failure to obtain a net increase after a given number of iterations), our final results might be improved. This possibility should be examined further.

An example of the method first described, using four iterations, is shown in figure 2-42. The links chosen as link i in step 2 are indicated with x's, and the direction of the vector, V_s , is indicated by the arrow under each layout.

A similar tack might be taken with the realization of radial and circular directional consistency. However, in these cases step 1 need not be performed, since the direction to be maximized is always predetermined, in other words, radially outward or inward, or circularly clockwise or counterclockwise. It is also necessary that, for the complete series of iterations, a fixed center be chosen with respect to which radial or circular movement is to be maximized.

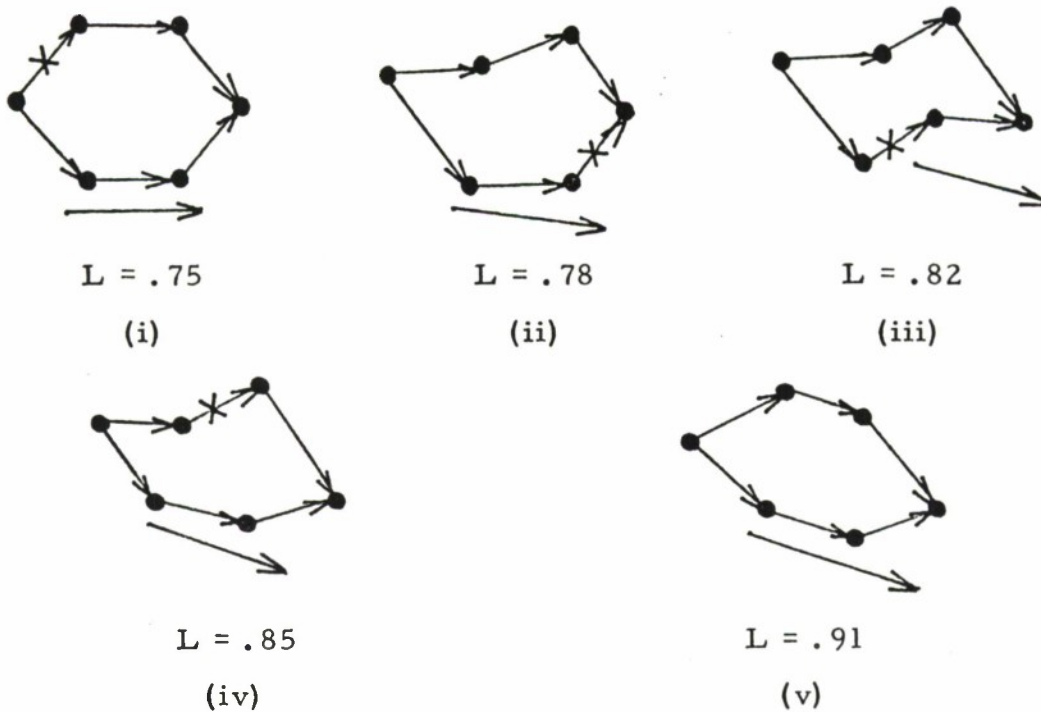


Figure 2-42

For example, optimization of the radial outward (or inward) measurement might proceed by several iterations of the following steps:

- 1) Find the link not yet tried for this iteration for which the product of the link length times the angle between the link and the radially outward (or inward) vector from (or to) the fixed center to (or from) the midpoint of the link is largest. Call this link i . If all links have been tried, the realization process is terminated.

- 2) Using the center of link i as a pivotal point, rotate the link until it lies in the direction of the radially outward (or inward) vector. If the resultant measurement of outward (or inward) radiality, R_o (or R_i) is not improved, or if the new node positions cause

overlap, return the link to its original position, and go back to step 1; otherwise, the new position is kept, and the iteration is complete.

Similarly, for circular movement, in step 1 we look for the link which deviates the most from the directed tangent at the link center point to a circle with the fixed center as its center. And in step 2, we rotate such a link until it is oriented in the direction of the tangent.

Examples of these two realization methods are shown in figures 2-43 and 2-44. Figure 2-43 shows a series of two iterations to improve R_o , and figure 2-44, to improve C_{\rightarrow} . Lozenges indicate the chosen fixed centers. The discussion of the problems relating to the method given for realization of linear directional consistency also applies here.

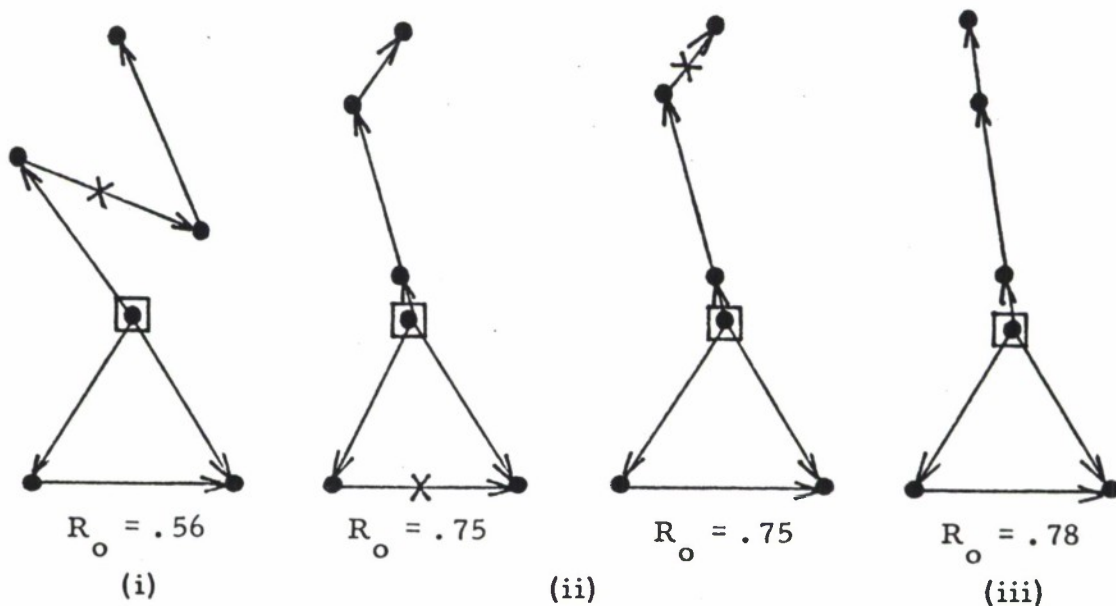


Figure 2-43

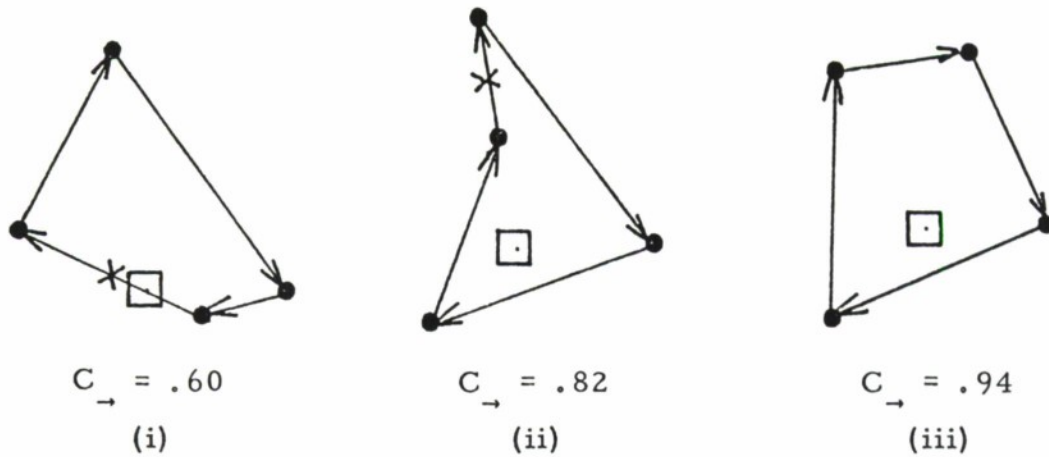


Figure 2-44

As a final note in the discussion of directional consistency we mention that several algorithms have been developed for the linear case. These methods, such as the one developed by Di Giulio and Tuan (12) which were intended for the layout of directed networks, attempt to place elements so that, if one element feeds another, the first lies consistently to one horizontal side of the second (say, to the left). The vertical coordinates are then chosen so that intersections are minimized. These algorithms are not actually concerned with the measure L , as we are, but the results tend to produce similar effects.

2.2.2.6 Minimum Number of Bends

The next quality to be considered for realization is the number of bends in a layout. As mentioned above, Fary has proven that every planar graph without self-loops and parallel links may be drawn without bends. A constructive realization algorithm may be

derived from his proof. This algorithm is recursive in that it finds a bend-free layout for the subparts of the original underlying graph before it can build the final layout for the whole graph. It is described and illustrated briefly below.* The steps mentioned are guaranteed to be possible as a result of Fary's proof. Given a layout, G , to obtain a straight line (bend-free) representation, $S(G)$:

A) Form a triangulated version of G , G' , and go to step B.

A triangulated version of a layout, G , is that layout with enough links added so that every region is bounded by exactly three links.

B) If G' has three nodes, form an equilateral triangle of the nodes, call it $S(G')$ and go to step C. If G' has more than three nodes, pick an interior node, n , of maximal degree, m , and label the nodes adjacent to it in clockwise order of links as n_1, \dots, n_m . If there is a node n_i , $2 < i < m$, such that $n_i n_1$ is in G' , perform (2) below; else, perform (1) below:

(1) Form G_1' from G' without node n and its links.

Find $S(G_1')$ (i. e., apply this algorithm to G_1'). $S(G')$ is then $S(G_1')$ with node n and its links placed inside the region bounded by the circuit containing links $n_1 n_2, n_2 n_3, \dots, n_m n_1$,

* Although the algorithm uses a layout as its input, this layout is essentially ignored in the resultant layout. Hence, the algorithm is considered to be a constructive realization algorithm.

so that no link adjacent to node n_i intersects with any of the links $n_i n_{i+1}$ ($i = 1, \dots, m - 1$); go to step C.

(2) Form two layouts:

(i) G'_{1_1} , including the cycle $n_1 n_i n_i n_1$ and all links and nodes inside this cycle in G' . Find $S(G'_{1_1})$ (i.e., apply this algorithm to G'_{1_1}).

(ii) G'_{1_2} , including the cycle $n_1 n_i n_i n_1$ and all links and nodes outside this cycle in G' . Find $S(G'_{1_2})$ (i.e., apply this algorithm to G'_{1_2}).

$S(G')$ is the result of placing the interior of $S(G'_{1_1})$ inside the region $n_1 n_i n_i n_1$ in $S(G'_{1_2})$. Go to step C.

C) $S(G)$ is $S(G')$ with the links added in step A removed.

Processing is then complete for $S(G)$.

An illustration is given in figure 2-45. Relevant steps are noted in parentheses. The superscript stars act as left parentheses and the subscript stars as right parentheses in the recursion.

This algorithm works well with those layouts whose underlying graphs are planar, in the case that the user prefers a constructive algorithm. However, it seems wise to also explore the possibility of defining a modifying algorithm which may be applied to all layouts. Let us try to describe such an algorithm.

For each bend point in the layout, in turn, examine the two points (either nodes or other bend points) adjacent to this bend point.

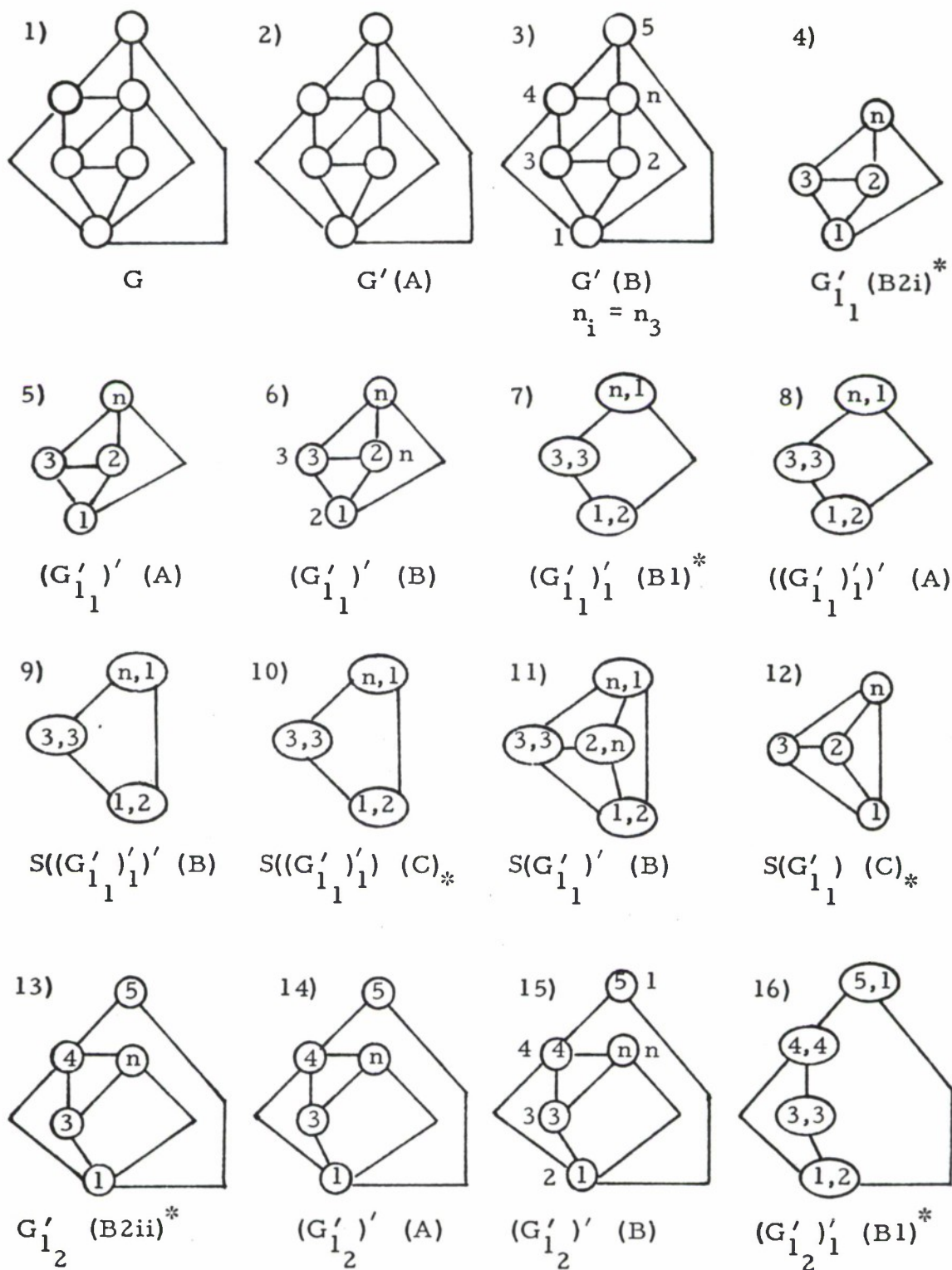


Figure 2-45

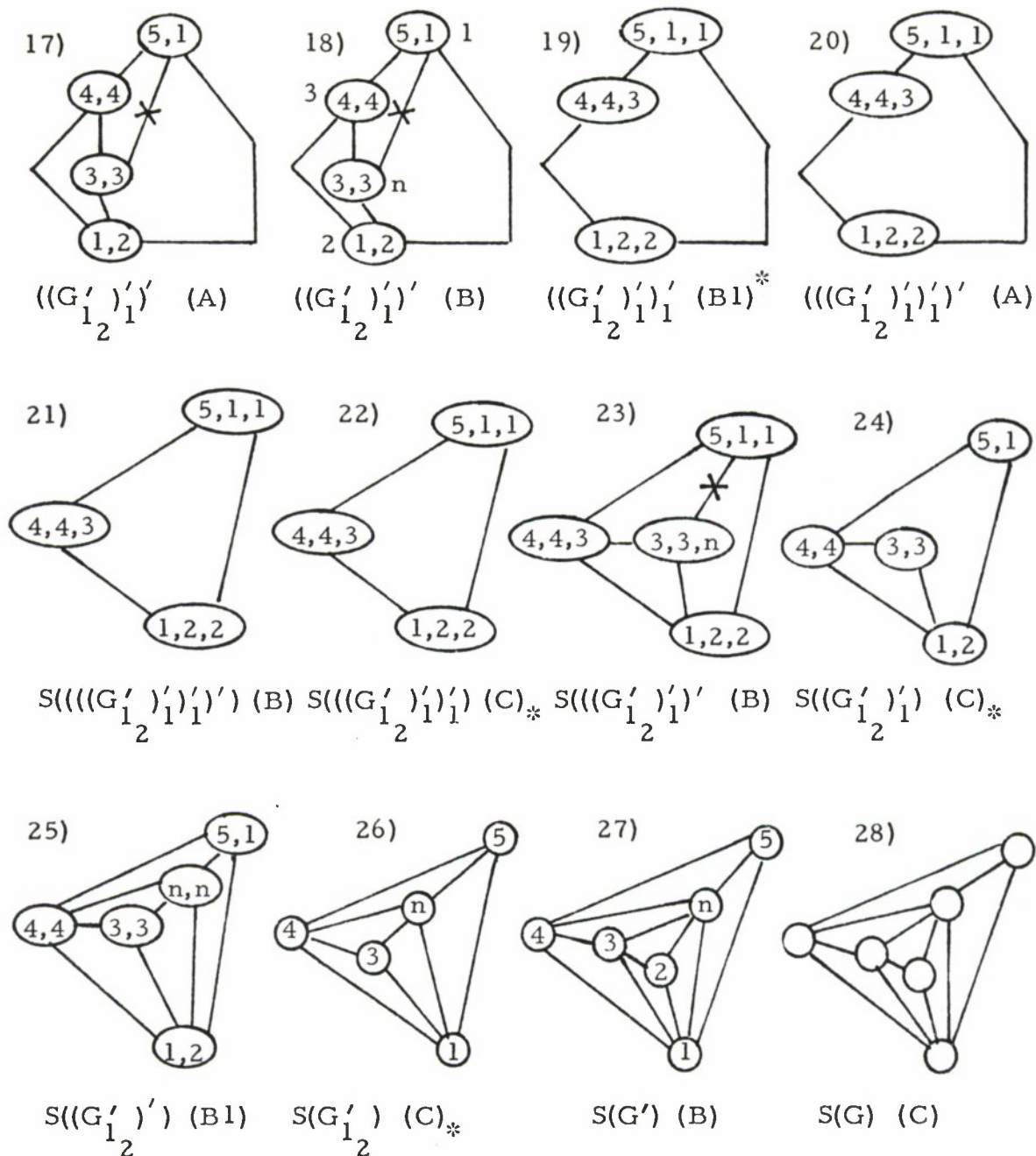


Figure 2-45 (continued)

Call these adjacent points a and b . If a straight line can be drawn from a to b without intersecting any link segment or node, it is drawn, and this bend point is removed. However, if some obstruction exists between a and b , it must be removed, before we can

draw a straight line between a and b to remove this bend.

It is here that we run into difficulty. The removal of such an obstruction involves a modification of node (and possibly other bend point) positions of the layout, and may be done in several ways. We may somehow move one or both of the two points, a and b, or we may move other points from one side of the line a,b to the other in some manner. The problem here is that for different cases, the manner of modification should be different, and it is difficult to determine automatically what type of movement is appropriate in a given case.

For example, in figure 2-46a, it is quite clear that a and b should be moved to the left, whereas, in figures 2-46b and 2-46c, we should move nodes other than a and b, in one case to the right

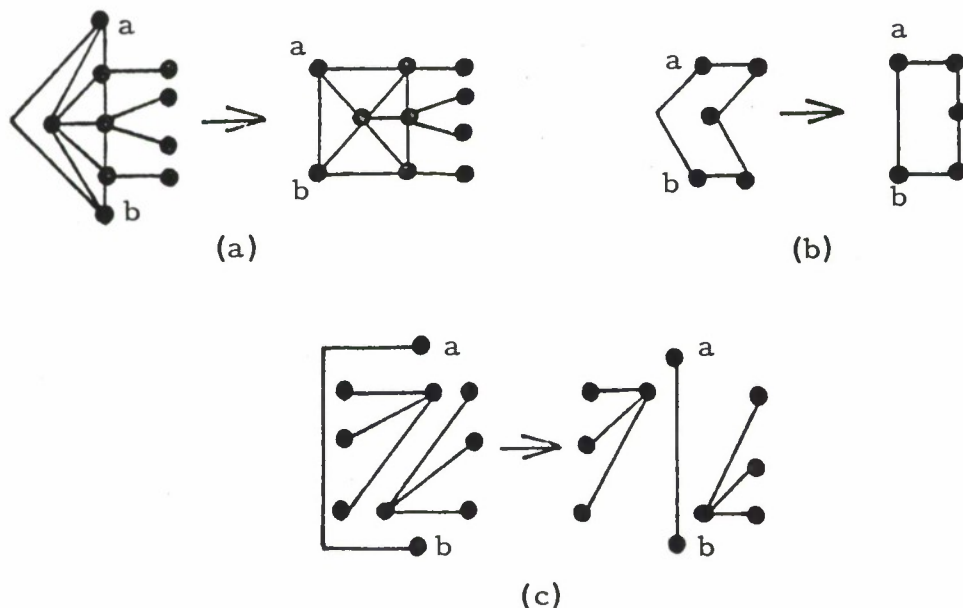


Figure 2-46

of (a,b), and, in the other, both ways. Thus we have the dilemma of determining for each case, what type of modification is best suited to the layout. Furthermore, it is not even clear that we should treat bends one at a time. In 2-46c, for example, we have similar conditions for both bends, and processing them simultaneously would be wise.

This discussion leads us to the conclusion that, in fact, the best modifying approach to bend point removal is manual, rather than automatic. This is supported by the fact that, unlike realization processes for most other qualities, that for bend removal is quite easy for the user to accomplish manually. There is no measure to be taken except the counting of bends, and all effects are visual. In anticipation of this conclusion, the MOD system, as will be seen in section 2.3, has been designed so that addition and removal of bend points is a simple operation. Thus, again we choose to depend on the interactive nature of the environment in which we are working, rather than developing a complex algorithm to perform a relatively simple task.

2.2.2.7 Minimum Number of Intersections

The next problem to be discussed, the minimization of the number of intersections in a layout, is one which has been studied for many years and is considered to be a key problem in the

automatic generation of printed circuit layouts. As a result, many constructive algorithms have been proposed, as well as what we shall call semi-constructive algorithms, those for which node positions are prespecified, and link paths are to be generated.

Underlying many of these semi-constructive algorithms we find the basic idea developed by Lee (24), of finding paths for wires (links) given element (node) positions and the positions of other wires. This approach performs a search which finds the optimal path according to any criterion prespecified by the user. With this method paths are found for the wires one at a time. Breuer (9) notes that the order in which wires are placed with this method affects the results and layout. Vincent-Carrefaur (37) skirts this problem by proposing that all wire paths be generated simultaneously. In section 3.2.2.5 these ideas will be elaborated further, along with other methods for minimizing intersections which were developed for the purpose of automatic circuit layout.

However, we would now like to consider a modifying algorithm aimed at achieving fewer intersections in a layout, while changing as little as possible. A seven-step algorithm which accomplishes this is given below. The main idea of the algorithm is to move nodes from one region to another in order to minimize the number of intersections. Nodes are considered one at a time, and are placed in a region of the layout which minimizes the number of intersections in

which the links adjacent to that node are involved. Once a node has been moved in such a way that the number of intersections is reduced, the algorithm is restarted. If all nodes are tried and no improvement found, the algorithm terminates.

A large part of the algorithm is concerned with determining the order in which nodes should be considered. Nodes which involve the largest number of intersections are considered first. Initially we only allow moves which remove all the intersections a node is involved in. If no such move is possible for any of the nodes, we then try moves which remove all but one intersection for a node. If this fails, we allow two intersections, and so forth. This process is controlled with a counter, I , in the algorithm.

The algorithm is then as follows, given a layout, L :

1) For each node n , find the number of intersections there are involving links adjacent to n , and call this number I_n . Call the degree of node n , D_n . Go to step 2.

2) Form an ordered list of nodes as follows:

If $I_{n_i} > I_{n_j}$ then n_i precedes n_j . If $I_{n_i} = I_{n_j}$ and $D_{n_i} < D_{n_j}$, then n_i precedes n_j . If $I_{n_i} = I_{n_j}$ and $D_{n_i} = D_{n_j}$, then order is arbitrary for the two. Call this list NODES. Set $I = 0$, and go to step 3.

3) Remove all nodes, n_i , from NODES for which $I_{n_i} \leq I$.

Go to step 4.

4) If NODES is empty, terminate the algorithm with L as the resultant layout; otherwise, mark all nodes in NODES unprocessed, and go to step 5.

5) If all nodes in NODES have been processed, add one to I and go to step 3; otherwise, go to step 6.

6) Take the first unprocessed node in the list NODES, call it n_i and mark it as processed. Form a set called CONNECT of nodes to which n_i is adjacent. Remove n_i and all connected links from L . Make temporary nodes of all remaining points of intersection, and call this new layout L' . Go to step 7.

7) (a) Look for a region R of L' (the infinite region should also be considered, i.e. that surrounding the whole layout), for which $D_{n_i} - 1$ nodes of CONNECT lie on its boundary, and for which if node n_i were placed inside R , each of the nodes of CONNECT, p , not on the boundary, may be joined to n_i with exactly one intersection resulting from the connection (i.e. they lie on the boundaries of regions adjacent to the region, R).

(b) If such an R is not found, restore the layout to L and go to step 5. If such an R is found, remove the temporary nodes, place n_i in R , and draw the links as specified above (bends are allowed). The result is the new layout L . Go to step 1.

A few comments on the algorithm will be helpful. Each time the algorithm returns to step 1, a reduction has been made in the number of intersections. Since there is a lower limit for this number for every graph, the algorithm always terminates. As mentioned above, the counter I indicates how many intersections are allowed in finding a new position for the node n_i under consideration in step 6-7. In the first iteration of a pass through the algorithm I is zero, indicating that we will only move a node to a new position when a position can be found which makes all the links of the node intersection free. If no position can be found for any of the nodes under this condition, we have another iteration in which we allow positions which remove all but one intersection for a node, etc. When I is large enough so that we are removing no intersections for a given node, we no longer consider that node (step 3). When no nodes remain to be considered, due to the size of I , the process is terminated.

The question remains as to whether or not this process finds the minimum number of intersections for the graph underlying the given layout. We make no attempt here to prove or even to claim such results from this algorithm. Such a claim would be extremely difficult to prove, since, even at the present time, expressions and methods to find such minima are quite complex or non-existent. We can only guarantee that for each pass through the algorithm either the

number of intersections is reduced or the algorithm terminates.*

The example shown in figure 2-47 illustrates the process. Each line is to be read from left to right. The pairs (I_n, D_n) are shown adjacent to the nodes where appropriate. Single numbers give positions in the list NODES. Nodes drawn as empty circles indicate that they are members of CONNECT, and nodes shown as lozenges are temporary nodes. Regions which satisfy the conditions in step 7 are indicated by R's. New layouts are shown only when changes warrant it. When a node number is slashed it has been processed, otherwise it is considered unprocessed.

* An alternative and more complex procedure exists for step 7, which produces more possibilities for the placement of a node n_i with D_{n_i} links, and I intersections. Some of these possibilities are not considered in the algorithm as written, but may turn out to be necessary for guaranteeing a minimal result. In order to include these possibilities, step 7a would proceed as follows:

(7a) Look for a region R of L' (the infinite region included) for which if we placed n_i in R , the integers a_0, \dots, a_I satisfy the following conditions:

$$1) \quad \sum_{k=0}^I a_k = D_{n_i}$$

$$2) \quad \sum_{k=0}^I k a_k = I$$

3) a_0 nodes of CONNECT lie on the boundary of R_i

4) For each of a_k ($k=1, \dots, I$) nodes of connect, n , a minimum of k intersections are required to connect n_i and n .

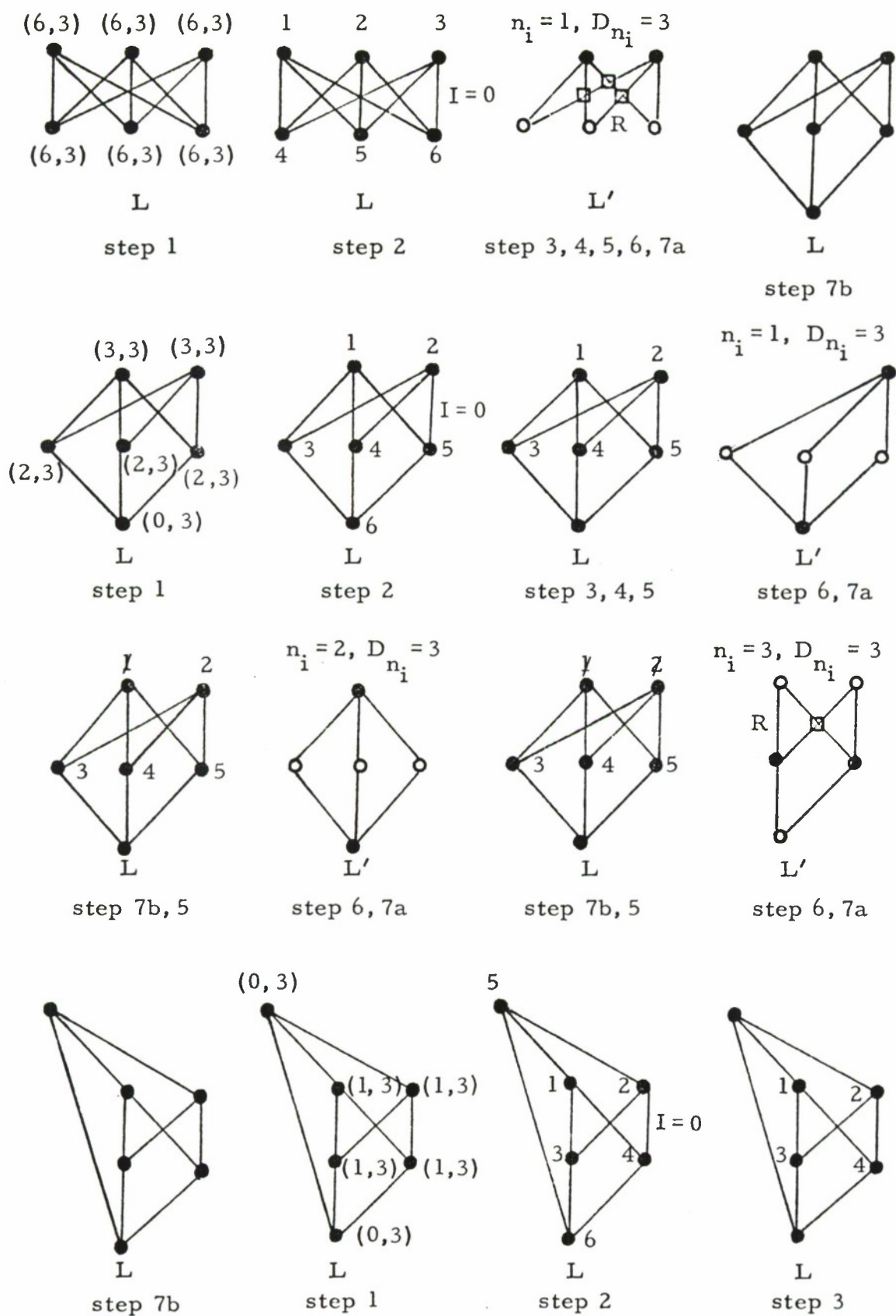
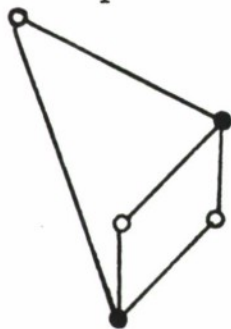


Figure 2-47

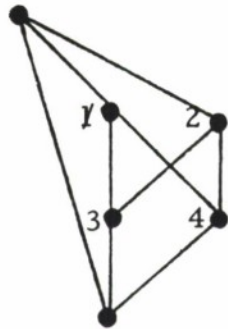
$n_i = 1, D_{n_i} = 3$



L'

step 4, 5, 6, 7a

$n_i = 2, D_{n_i} = 3$



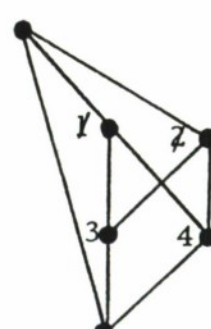
L

step 7b, 5



L'

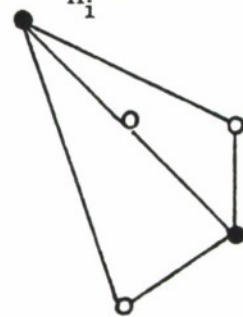
step 6, 7a



L

step 7b, 5

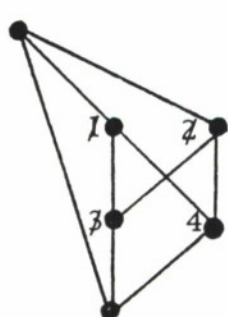
$n_i = 3, D_{n_i} = 3$



L'

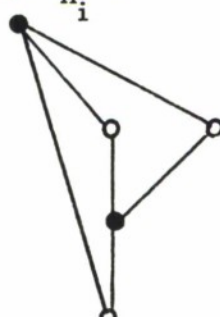
step 6, 7a

$n_i = 4, D_{n_i} = 3$



L

step 7b, 5



L'

step 6, 7a



L

step 7b, 5

$I = 1$



L

step 3, 4

Figure 2-47 (continued)

2.2.2.8 Minimum Link Length

The problem of decreasing total link length has also been of concern to the developers of methods for automatic circuit layout. Thus many algorithms, both constructive and modifying have been devised and are discussed in section 3.2.2.5. Among them Lee's algorithm mentioned above as a semi-constructive algorithm may be used for this purpose.

However, another algorithm, developed by Steinberg (33) meets the requirements of a modifying algorithm for minimization of link length quite well. The details of this algorithm are given in section 3.2.2.5. In summary, Steinberg's method finds a group of unconnected nodes, removes them from the layout, and then repositions these nodes in a manner which minimizes the total length of the links to which they are adjacent. Each set of unconnected nodes is processed, in turn, in this manner until no more improvement can be made. Variations and improvements on this scheme have been considered by Rutman (32).

2.2.2.9 Parallelism

Next we consider methods to minimize the number of different link slopes in a layout (or to increase parallelism). One trivial solution might be to redraw all the links as series of horizontal and vertical segments, wherever possible. This method would guarantee

to yield the minimum number of slopes for a layout, but it may require the introduction of several bend points, and does not conform with our idea of changing the layout as little as possible.

A more reasonable modifying algorithm is given in Appendix 4. It is not included in the text due to its length. The idea of this algorithm is to first place all of the link segments into sets. Once these sets of link segments have been formed, the link segments in each set are adjusted so that they are parallel to one another where possible. As the algorithm is now written, originally parallel link segments remain parallel. Again, the algorithm appears in Appendix 4, along with a discussion, and an example of its use.

2.2.2.10 Horizontal-Vertical Orientation

The final quality to be considered for realization is that of horizontal-vertical link segment orientation. A method which accomplishes this by reorientation of the links in these two directions has been implemented in the MOD system and is described in section 2.3.3 as a series of two commands, merge c and merge r.

In summary, we have considered the measurement and realization of many of the qualities discussed in section 2.1. Realization algorithms were developed which, for the most part, are intended for implementation in an interactive graph building and layout environment such as that described in the next section. Discussion of

algorithms for a few of the qualities was delayed until section 2.3.3, since these algorithms have already been implemented in the MOD output system.

2.3 THE MOD SYSTEM

The MOD system was developed to provide an interactive graphics environment for experimentation with various layout types and layout algorithms. It has been implemented on a PDP-1 with 6k of 18-bit core along with a drum, a teletype, and paper tape I/O. Peripheral equipment used included a typewriter, a dec tape unit, a Calcomp plotter, a refreshing CRT, and a Rand tablet.

The system was designed in three parts: Mod Input, Mod Framemaker, and Mod Output. Mod Input allows the user to draw and modify graph layouts of several types, to store these on paper tape, and plot them on a Calcomp plotter. Graph layouts previously drawn and stored on paper tape may be read into MOD Input and processed again. The MOD Framemaker is similar to MOD Input, except that no Calcomp output is possible. Instead, a facility has been provided with which the user may store sequences of graph layouts (frames) on dec tape or temporarily on the drum. These sequences may later be replayed or modified through the Framemaker. The motivation for this system was to provide a facility for making films or videotapes using the results of the rest of the MOD system.

The third part, MOD Output, was intended for experimentation with layout algorithms such as those described in section 2.2.2. Graph layouts may be read in on paper tape, but their structure may not be modified. The only changes allowed are those involving the layout of the given graph. Again, the output may be either paper tape or Calcomp plotter.

The following three subsections describe in more detail how these three systems are used. Further details on the structure of the system may be found in Appendix 5. The last of the subsections also includes a more detailed discussion of the layout algorithms implemented in the MOD system at the present time.

2.3.1 The Input System

The MOD Input system provides a means for inputting various types of graph layouts. The user first selects the kind of graph layout he wishes to draw, and then proceeds to draw graph layouts of this type. He may output the graph layouts either on paper tape or on the Calcomp plotter.

Upon starting, the system types out the options from which a user must select a graph layout type. The options are:

1. a) directed
b) undirected
2. a) net
b) graph
3. a) adp's
b) none

4. a) ep's
 b) none
5. a) shapes
 b) standard
6. a) define shapes
 b) no
7. a) functions
 b) none

Option 1 is clear. Option 2 allows the user to specify that either

- a) links may branch, or
- b) links may not branch.

Option 3 specifies that the nodes in the graph layouts to be drawn may include, in their definition, certain points from which links or arcs may originate (arc departure points, ADP's). Normally, a link leaves a node from its center point, with that portion of the link which lies inside the node removed from the graph layout. When ADP's are specified for a node, the user may draw a link either from an ADP or from the center of the node. Option 4 provides for arc entry points (EP's) which are similar in nature to ADP's, except that they specify points on nodes at which links may terminate. Again, the center point may be used for termination.

Option 5 allows the user to choose between using one standard node shape, provided by the system (a square), or several shapes, either defined by the user or read in on paper tape (N.B., this last facility has not yet been implemented). Option 6 specifies whether

the user is to define these shapes or read them in. Option 7 provides the user the ability when he is defining shapes to associate a number with each shape he defines, which will be stored with that shape.

Note that 6a may only be used if 5a was specified. The sequence 5a, 6b implies a facility not yet available. Option 7a may only be chosen if 6a was chosen.

Upon completion of the option specification, the main frame for MOD Input appears on the scope. From this point on, the system is controlled, for the most part, with the Rand tablet. The main frame is shown in figure 2-48. Pen position is shown by a small "+" on the scope.

If the user has chosen to use the standard node shape, this shape will appear at the top of the right hand column. In this case the words "page" and "define" will not appear in the menu at the bottom of the main frame. On the other hand, if he chooses to define node shapes, those he has defined will appear in this column. If there are too many shapes to fit into this space, pointing to "page" will cause another group of defined shapes to appear.

At this point in the use of MOD there is a current shape. Initially, this is null. To make a shape current, the user must point to its prototype in the right hand column.

To define shapes, the user points to "define." The main frame is replaced by the define frame (see figure 2-49). All shapes

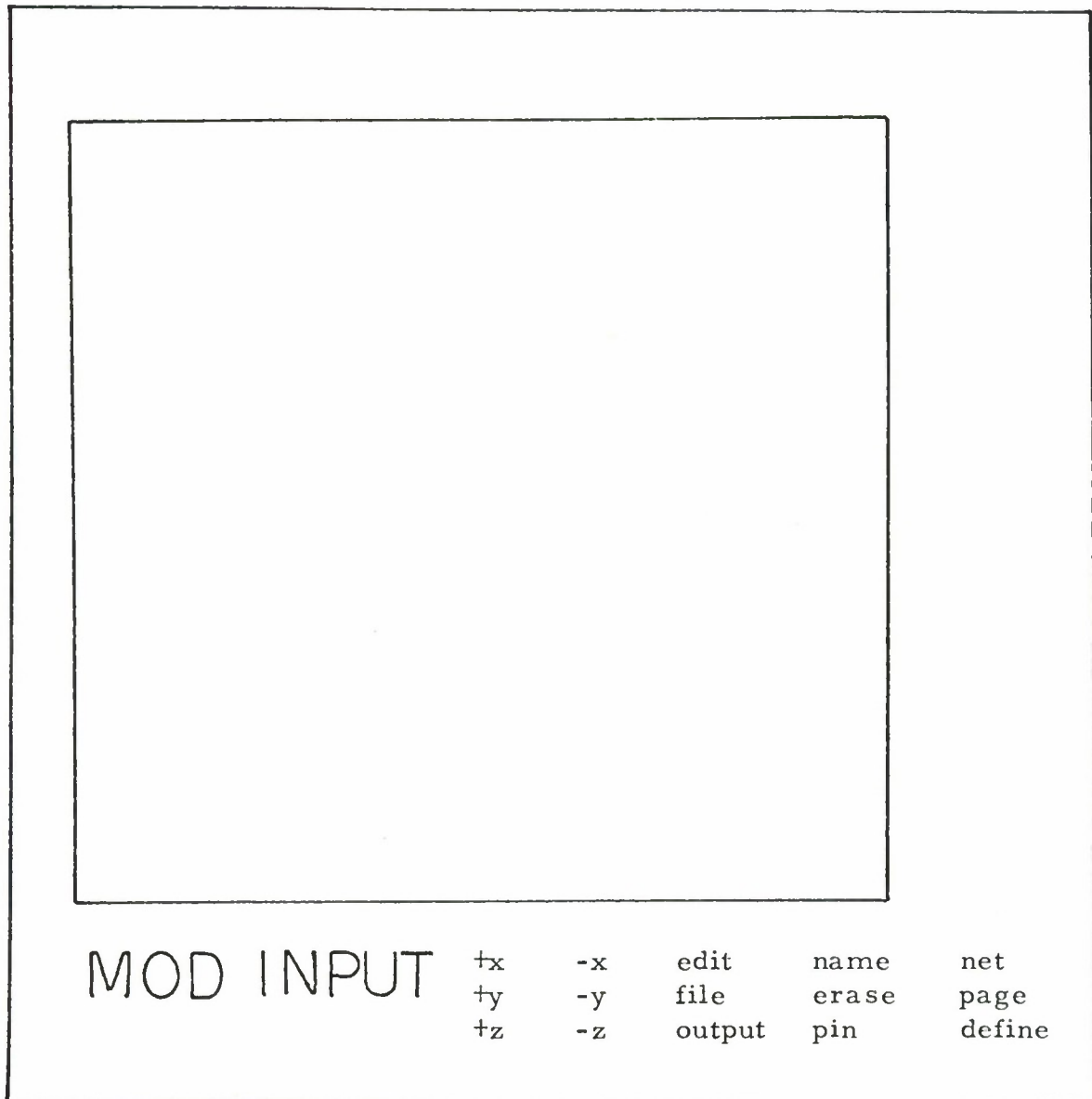


Figure 2-48

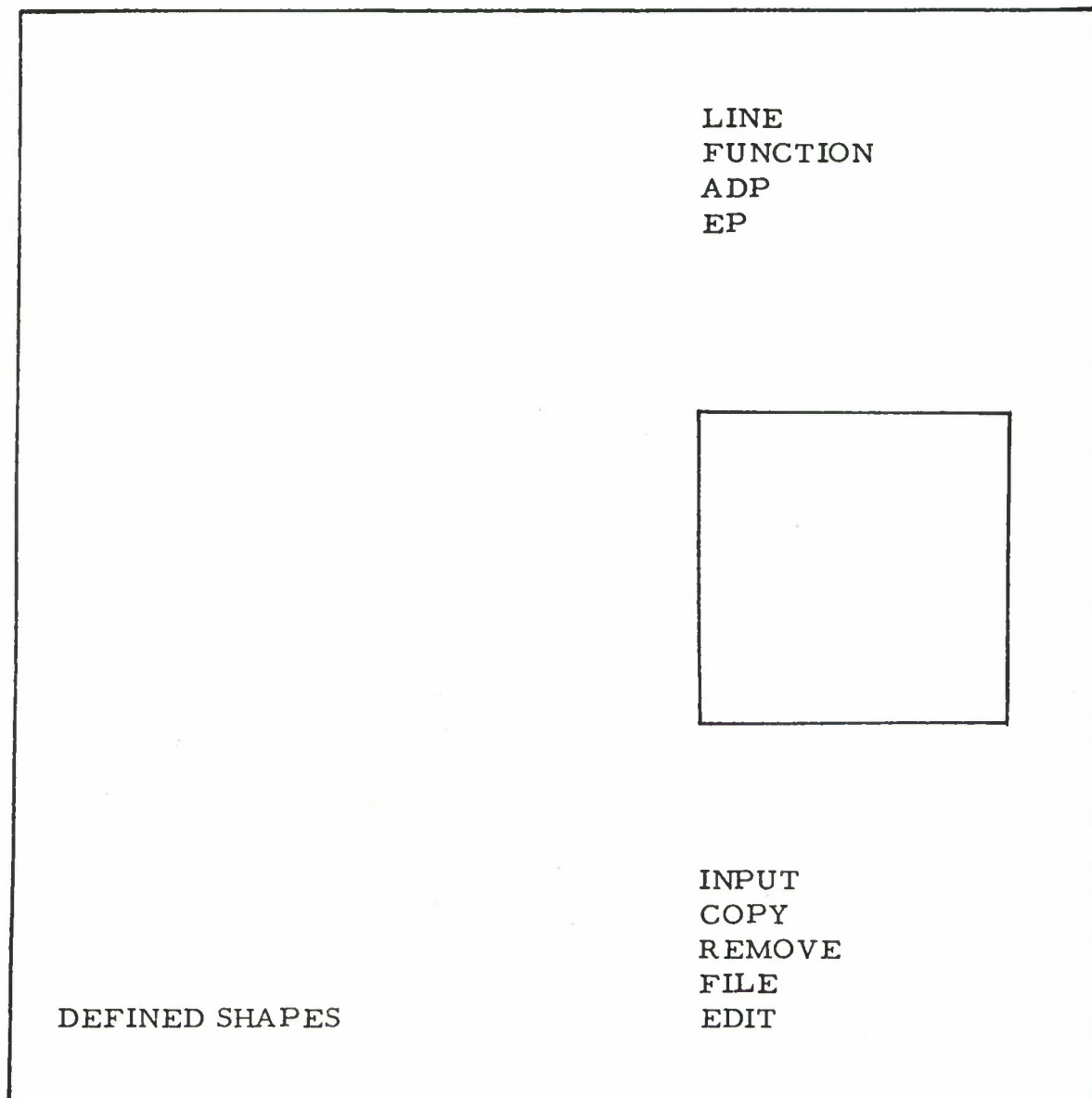


Figure 2-49

already defined and filed appear to the left in this frame. To define a new shape, the user first points to "line." As long as "line" remains illuminated, he may draw lines in the box on the right of the define frame, and these will be recorded as part of his shape definition. Each line must begin and end inside the box. To begin a line the pen is pressed down; the line ends when the pen is lifted. If a line is begun inside the box but the pen is taken out of the box before the line is completed, no line is remembered, but the word "line" remains illuminated.

If functions have been specified (Option 7) the word "function" appears in the define frame menu. Pointing to function causes the typewriter to output "type two numbers." The user must respond by typing two digits between 0 and 7, which then become the function name for the shape he is defining. He may change the function name by simply repeating this procedure. Initially the function name for a shape is "00."

If ADP's (Option 3) have been specified, the word "adp" appears. Pointing to "adp" causes this word to illuminate. The user may then put the pen down any place within the box. Each time he puts the pen down, he specifies another ADP for the shape he is defining. He may include as many as he likes, or none. For each ADP specified, a small "x" appears in the shape.

Similarly, when EP's (Option 4) are chosen, the word "ep"

appears in the define frame. "Ep" functions in the same manner as "adp" except that for each point specified, a small "o" appears which functions as an EP for the shape.

To turn off "line," "adp," or "ep," the user simply puts the pen down anywhere outside the small box (for example, on any other label). When no label is illuminated, the user may point inside the box to erase part of its contents. This is called scrubbing. To erase ADP's or EP's he presses the pen down on these points; to erase lines, he points to their beginning points.

When a shape definition is completed, it must be filed to be remembered. This is done by pointing to the word "file." The shape in the box, along with its function, if it has one, will then appear in the list of defined shapes on the left.

To erase the contents of the box completely, whether or not the shape has been filed, the user points to "remove." It is important that he remember to file a shape he wishes to retain, before removing it.

To change or remove a shape which has already been filed, the user first points to "edit," which illuminates, and then to the shape on the left he wishes to edit. Pointing elsewhere will simply turn off the edit light. The shape then appears inside the box and is removed from the list of defined shapes. Any changes may be made to the shape by using "line," "adp," "ep," "function," "copy" (to be

explained below), or by scrubbing. When the edited shape is refiled, each occurrence of the shape is changed in the graph layout of the main frame, if any appear there, but all links to and from any occurrence of this shape are removed. However, if, when editing a shape, "remove" is pressed, the shape edited is considered deleted and will totally disappear from the layout in the main frame.

Any filed shape may be copied for modification and filing as another shape definition. This is done by pressing "copy," which illuminates, and then pointing to the defined shape to be copied. Pointing elsewhere will simply turn off the copy light. A copy of the specified shape will appear in the box. The user may then continue any way he likes.

When all defining is done, the user presses "input" to retrieve the main frame. All shapes he has defined and filed, or edited and filed will appear in the right hand column along with old, unedited shapes. He may define shapes again at any point during the session.

A graph layout is drawn inside the box in the main frame. Nodes are obtained by drawing an x at the position desired. A node with the current shape will appear. Links consist of a series of straight line segments. The points at which these segments connect to one another in a link are called pins (or bends). To draw a link, the user puts the pen down in the center of a node or at an ADP, wherever he wishes the link to begin; then, keeping the pen down, he

draws the first line segment, and lifts the pen up. If he ends at a node (or EP) the link is considered completed. Otherwise, he must draw the next line segment of the link beginning at the end of the last line segment. This process continues until he ends a segment at a node or EP. A link of one segment cannot begin and end at the same node. Any incorrectly drawn links will disappear immediately.

Should the user wish to copy or move any portion of the graph layout he has drawn, he may draw a closed shape around that portion. A closed shape must end close to the point at which it began. To move that part of the graph layout, he then points inside the closed shape. By keeping the pen down he maintains a handle on the enclosed portion of the graph layout, and, as he moves the pen, this portion will move with it. When the pen is lifted, the move operations ends. Any nodes and pins enclosed may be moved in this manner.

To copy a portion of the graph layout, the user again encircles that part, but then points outside the closed shape, keeping the pen down. The copy soon appears, and, as in the move operation, as long as the pen is down, the user may move the copy anywhere in the box. Upon lifting the pen, the copy remains stationary. The copy of the portion of the graph layout enclosed contains all nodes and pins of that portion (but not their names), and all link segments whose beginning and end points are on nodes or pins in that portion.

The user may also remove an element of the graph layout by

scrubbing it. A scrub consists of drawing in an erasure motion on the object to be scrubbed. Scrubbing a node causes the node and all attached links to disappear. To remove a link, its visible origin must be scrubbed. When a pin is scrubbed, it disappears, and the two line segments it connects become one.

It must be noted that with the five previous operations, all of which are initiated by drawing instead of pointing, the pen movement is "inked" (traced with small dots). When the operation is recognized by the system, the inking disappears. If the effects of the operation are not seen reasonably quickly, it means that the pen movement was incorrect and did not initiate the operation which was intended. The operation must then be repeated.

The whole graph layout may be moved to the right, left, up, or down by pressing "+x, " "-x, " "+y, " or "-y, " respectively. The graph layout may be enlarged by pressing "+z" or made smaller by pressing "-z. " Nodes may be made only as small as their size when they were originally defined.

The number of pins in a link may be increased by going into pin mode. This is accomplished by pressing "pin. " When in pin mode, the user may point to any existing pin, or any link beginning. Keeping the pen down, he then has a handle on a newly created pin for this link, which he may move around as long as he keeps the pen down. This may be done any number of times in pin mode. To leave

pin mode, the user need only put the pen down anywhere else outside the square of the main frame.

When "name" is pressed, the user is in name mode. Any node he points to will cause the typewriter to carriage return. The user must respond by typing a six or fewer character name made up of alphabets, numbers, space, "+", and "-"; he indicates the end of a name with a carriage return. The name will then appear on this node. To leave the name mode, again the user points anywhere outside the square. The user may make the name characters larger by keeping sense switch 3 up.

To erase the whole graph layout, the user may press "erase." At this point the typewriter outputs:

New options

- a) yes
- b) no

If the user then types "a," MOD Input is reinitialized. If he types "b," the typewriter outputs:

Keep shapes

- a) yes
- b) no

If the user then types "a," all the defined shapes as well as the graph layout are removed, otherwise only the graph layout disappears.

If the user has specified net (Option 2) then the word "net" appears in the main frame menu. "Net" is pressed and used as if

it were a defined shape. However, nodes produced when net is the current shape (net nodes) have some special properties. They only appear when net is the current shape, and then they appear as small triangles. When directed links are used, no arrowhead appears on links ending at net nodes. These are the only such nodes. Net nodes may have no ADP's or EP's. Net may be removed from current shape status by simply pointing anywhere in the right hand column above the word "net." At this point all the triangles for net nodes disappear, although the nodes still exist. The intent of including net nodes is to allow for drawing links which branch. Net nodes provide an intermediary point for branching. Net nodes may also be used for placing labels on graphs in a convenient manner, since net nodes may be given names, and the names will remain even when the net node is not visible.

Three other labels appear in the main frame menu. These are used for input and output of graph layouts. Pressing "output" initiates the Calcomp drawing of the graph in the box of the main frame, and a drawing of the list of defined shapes if there are any.

Pressing "file" initiates the output of the graph layout, present options, and defined shapes on a paper tape. This is in a form which may be input later to any of the three MOD systems. To input such a tape in MOD Input, the tape must be loaded in the paper-tape reader, the reader turned on, and "edit" pressed. When a tape is

read in, any previous graph layout, defined shapes, and options are lost, and those on the tape become current.

2.3.2 The Framemaker System

The MOD Framemaker is a variation of MOD Input which allows the storing, editing, and playing of sequences of graph layouts (referred to now as frames). The only difference in the two systems is in the output command. Output no longer initiates Calcomp drawing; its new effects are described below. The main frame for the Framemaker is the same as figure 2-48.

Sequences of frames may now be created on the drum, and quickly replayed on the scope. These sequences may be output on dectape and saved for later read in to the drum.

To initiate any frame operation the user points to "output" on the scope. A carriage return will be typed. Then, any of the following may be typed, ending with a carriage return:

"id" - initializes the drum for storage of frames. Before any frames may be stored or retrieved, either the drum must be initialized, or, a dectape must be read in which has been previously output from the Framemaker. The current frame pointer (CFP) is initialized to block zero.

"td" - reads a previously output dectape onto the drum and sets the CFP to the first frame of this tape, displaying this frame.

If there are no frames in the sequence, the CFP points to block zero.

"dt" - outputs the drum frames onto tape, renumbering the frames so that they are numbered in their sequential order. This operation should not be attempted until the drum has been either id'd or td'd.

"m(-)xxx" - moves the CFP up or down in the sequence xxx (octal) frames and displays the new current frame. If xxx is large enough to go beyond the limits of the sequence, the last or first frame of the sequence will become the current one, depending on the direction of the move.

"s xxx" - moves the CFP to block xxx_8 . There are 377_8 blocks numbered from 1 through 377_8 , and each frame occupies one block. If xxx is not a frame, an error message is printed out and the old CFP is kept. If xxx is a frame, it is displayed.

"i (xxx)" - records the graph layout now on the scope as a frame and inserts it in the frame sequence directly after that pointed by the CFP, or after that frame mentioned, if a number is given. The CFP is then updated to point to this new frame.

"d" - deletes from the sequence that frame pointed to by the CFP. The CFP is moved to the previous frame in the sequence, or, if no previous frame exists, to the next frame in the sequence.

"p" - starting with the frame pointed to by the CFP, succeeding frames of the sequence are placed on the screen for a period of time proportional to the number appearing in the test word on the console of the computer. This process is halted by setting sense switch 1 up or by reaching the end of the frame sequence. The test word may be changed at any time during this procedure. The CFP is updated to coincide with the picture appearing on the screen.

If the operations "m, " "s, " "d, " or "p" are attempted when CFP = 0, or when there are no frames, an error message is printed out and the operation has no effect. Printout also occurs when the drum is initializing or when the CFP is updated. In the first case, the word "initializing" is printed. In the second, the number of the current frame is printed. The printout of frame numbers may be suppressed by setting sense switch 2 on. It is important to remember that the picture which appears on the screen may not be the same as the current frame, for, the user may modify what appears on the screen.

2.3.3 The Output System

The purpose of the MOD Output system is to experiment with layout algorithms for graph layouts drawn with MOD Input or Frame-maker. Upon starting, the system displays the main frame shown in figure 2-50. The system allows commands which change the layout

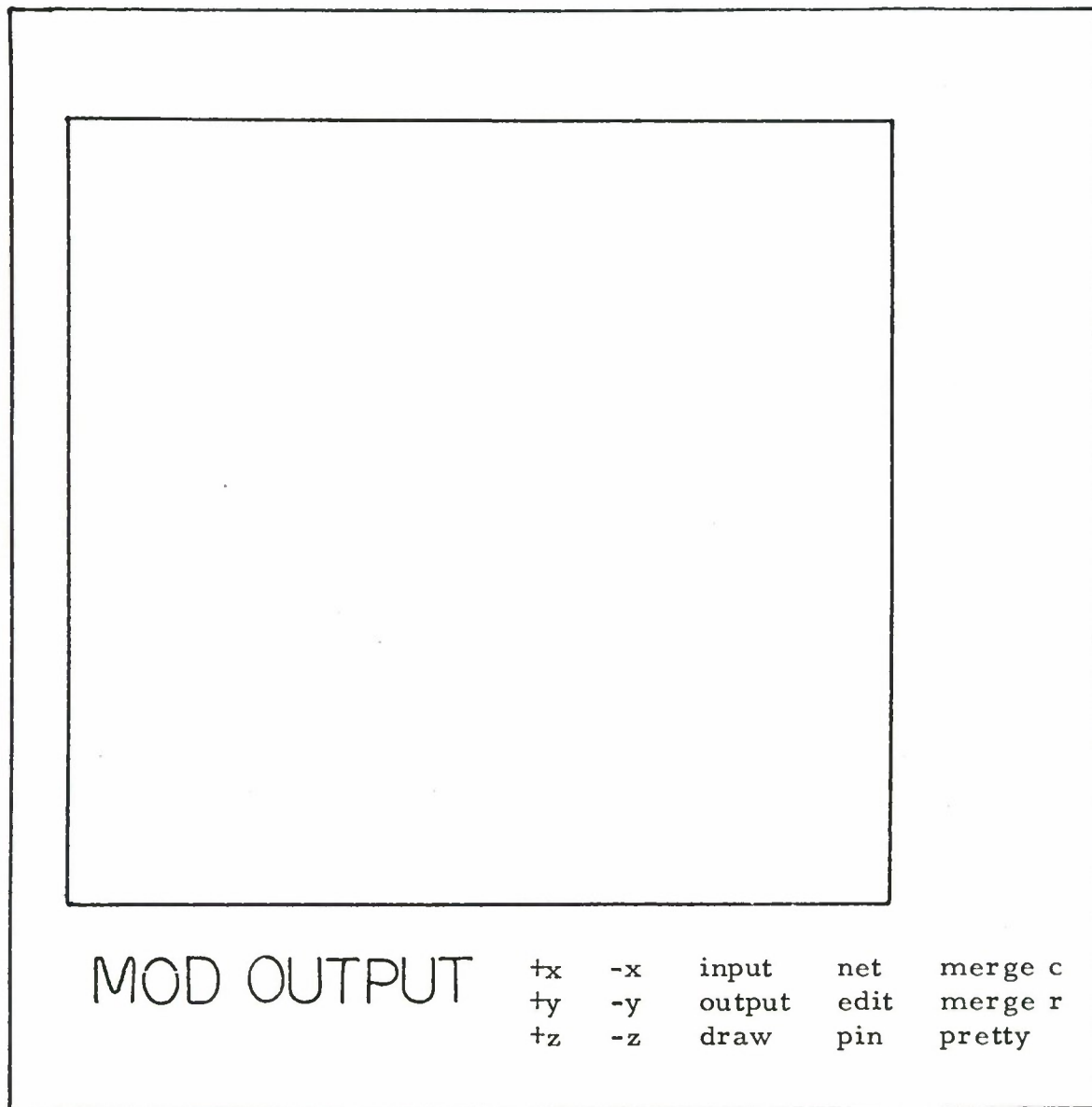


Figure 2-50

of graphs, but excludes those which change the content of the graphs underlying the graph layouts. The following MOD Input commands are included in MOD Output, and function in the same way as for MOD Input:

- +x, -x
- +y, -y
- +z, -z
- input (same as edit)
- output (same as file)
- pin

The draw command is similar to the output command in MOD except that only the graph layout is drawn. The defined shapes are not drawn. Neither do the defined shapes appear in the right hand column of the main frame of MOD Output.

The net command has been modified and now only controls the appearance of net nodes. To turn net off in MOD Output one points at any empty space outside the main frame box. Note that in order to obtain a graph layout on the screen, the input command must be given and a paper tape read in.

The edit command has not yet been implemented. It is intended to cause the read in of editing commands (perhaps on paper tape) for modification of the contents of the graph underlying the layout appearing on the screen. The form and function of these commands has yet to be decided upon. The remainder of the commands to MOD Output are intended for layout improvement. At present

there are three operations for this purpose, pretty, merge c, and merge r. These provide some facility for improving a graph layout. It is hoped that this list will be expanded in the future.

The pretty command is used to align nodes into rows and columns as mentioned in section 2.2.2.2. It has several options.

Upon pressing 'pretty, ' the following is typed out:

```
pins?  
a) yes  
b) no
```

If the user desires pins to be included in the aligning, along with nodes, he types 'a, ' otherwise he types 'b. ' This is followed by:

```
net nodes?  
a) yes  
b) no
```

The user types 'a' if net nodes are to be included and 'b' if they are not. It is suggested that the user experiment first with aligning regular nodes only. The next message is:

```
sspace =
```

The user must type an octal number which is the distance in scope units (the scope is 1024^2) within which two rows or columns of nodes are considered to be the same and may thus be aligned into one. The user must respond with a positive octal number (20_8 or thereabouts is suggested) followed by a space. The next message is:

space rows?

- a) yes
- b) no

The user types "a" if he wishes the rows and columns formed to be equidistantly spaced and "b" otherwise. It is suggested that this not be done unless net nodes and pins are also aligned, since there is a great deal of node movement when rows and columns are spaced equidistantly. If his answer was "b" the option specification is ended. If he typed "a" the typewriter responds with:

rspace =

Again the user responds with a positive octal number ending with a space. This number represents the distance in scope units between rows and columns (100_8 is suggested). The pretty operation is then executed according to the options specified.

The algorithm used to accomplish the first part of the pretty operation, the aligning of nodes into rows and columns, proceeds by repeating the following four steps, once for x-coordinates and once for y-coordinates:

- 1) Make a list, POINTS, of all points; if pins or net points are to be processed, they should be included; otherwise, the list consists only of nodes. Go to step 2.

- 2) If POINTS is empty, terminate. Otherwise, choose an element of POINTS, call it N, and remove it from POINTS. Set the list SAME to null, and go to step 3.

3) If POINTS is null, go to step 4. Otherwise, look for an element M of POINTS whose z-coordinate (z is x or y depending on which coordinate is being processed) is within SSPACE of N's z-coordinate. If none is found, go to step 4. Otherwise, remove M from POINTS, add it to SAME, and go to step 3.

4) For each element of SAME, make the z-coordinate equal to that of N. Go to step 2.

The second part of the pretty operation, which is optional, that of equalizing the distance between rows and columns to RSPACE, is accomplished by proceeding through the following four steps, again, once for each coordinate:

1) Make a list of nodes (and pins and net points if appropriate), and call it POINTS. Set IND = 1, and go to step 2.

2) If POINTS is null, terminate. Otherwise, find that element, N, of POINTS with the smallest z-coordinate, Z, and remove it from POINTS. Go to step 3.

3) If IND = 1, set IND = 0, set Y equal to Z, and go to step 4. Otherwise, set Y equal to Y+RSPACE, and go to step 4.

4) For all elements P of POINTS whose z-coordinates are equal to Z, set the z-coordinate of P equal to Y, and remove P from POINTS. Set the z-coordinate of N equal to Y, and go to step 2.

The pretty process is straightforward, and needs little explanation. The only problem is that if a row of nodes, for example, is drawn with a large enough range of y-coordinate, pretty might separate the row into two distinct rows as shown in figure 2-51. However, this is easily remedied by a small manual movement of the nodes and a reapplication of pretty for exact placement.

The merge c and merge r operations are initiated by pointing to one of these labels in the menu. As mentioned in section 2.2.2.10, they increase horizontal and vertical link segment orientation in a layout. When merge c is pressed the links of the graph layout are made to appear as vertical as possible. The non-net nodes (and their attachment points) are considered immovable, but pins and net nodes, if desired, may be moved for this purpose. Adjacent columns of pins are merged to form vertical columns on which links run, when no conflict occurs, as long as they are within sspace of one another. The user must input sspace in response to the message:

sspace =

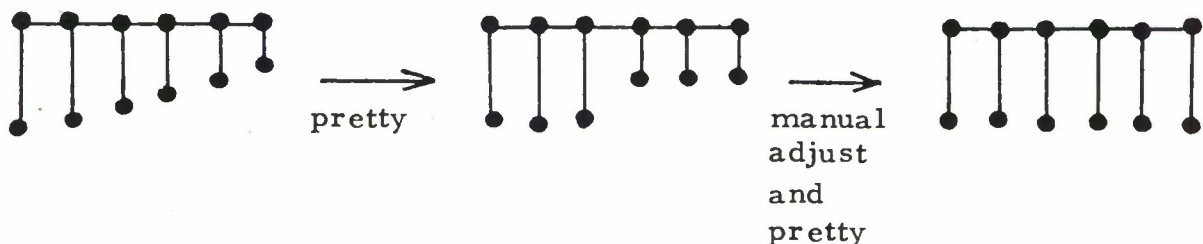


Figure 2-51

which is typed when merge c is activated. Again, the response must be a positive octal number (around 20_8 is suggested). This message is followed by:

net nodes?

- a) yes
- b) no

Responding with "a" allows net nodes to be moved along with pins.

Answering with "b" means that net nodes are considered immovable, as are all other nodes. The merge c operation is then executed.

Merge r proceeds in a manner similar to merge c. The only difference is that the aim is to make links as horizontal as possible. The final result of applying merge c and merge r should be a graph layout in which the link segments run horizontally and vertically (Manhattan geometry), wherever they can.

An explanation of merge r should suffice to make both operations clear. The merging process proceeds from top to bottom (and in the case of merge c, from right to left). Once we have determined that two rows can be merged, they are considered as one row. If a row, say a, cannot be merged with the row below it, no row below a is merged with a or any above a. The constraints which prevent rows from being merged are as follows, where NC stands for an immovable point (i. e. all nodes, and net nodes if they are not to be moved), and P stands for a movable point (i. e. all pins, and net nodes if they are to be moved):

1) Two adjacent rows cannot be merged if they both contain NC's.

2) Two adjacent rows cannot be merged if merging introduces overlap of NC's, P's, or link segments.

3) Two adjacent rows are not merged if merging changes the side of a link segment on which a point (NC or P) lies.

4) Two adjacent rows are not merged if they are more than sspace units apart.

Constraint one is clear, since NC's are not movable, and requires only a check on the contents of the two rows under consideration. Constraint two is broken up into several cases, each of which must be checked for. The cases are shown in figure 2-52 where the a's are elements of the top row being considered, and the c's are elements of the adjacent row. If any of these cases arises, the rows are not merged.

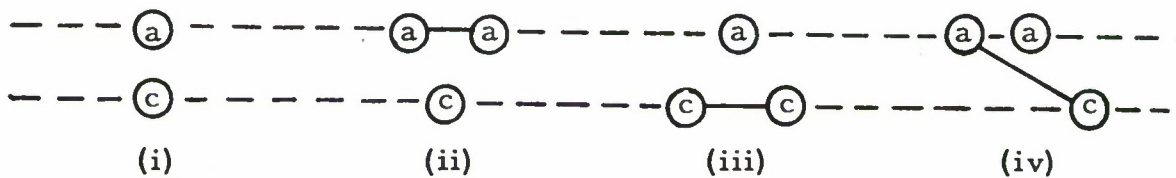


Figure 2-52

The third constraint is somewhat intuitive. There are two cases which must be watched for, as shown in figure 2-53. In 2-53a if the two rows were merged we would have the result shown in figure 2-54a. But if node a were connected to another node as shown



Figure 2-53

in 2-54b, then merging the two rows would cause a link intersection as in 2-54c. Similarly, in 2-53b if the two rows were merged we

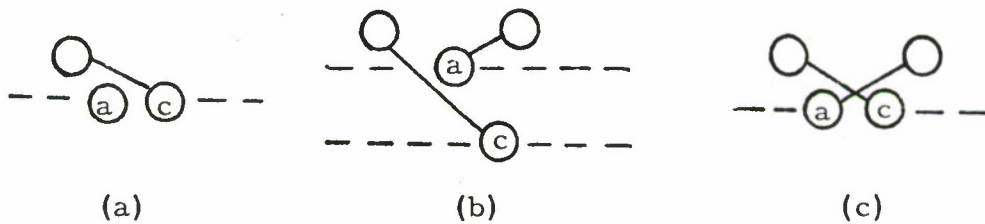


Figure 2-54

would have 2-55a; but if c were connected to another node as in 2-55b, then merging the two rows would cause a link intersection as

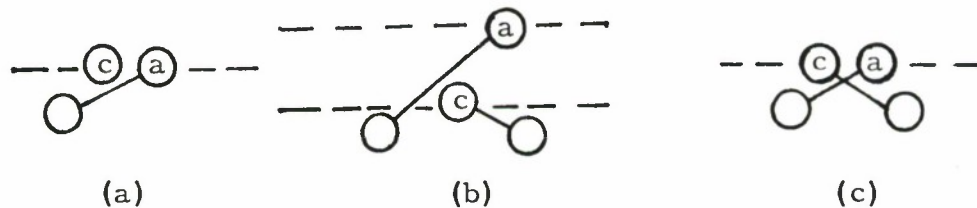


Figure 2-55

in 2-55c. Thus constraint three is observed in order to avoid the

possible creation of intersections.

Constraint four is straightforward, but it should be noted that if the top row, *a*, of two adjacent rows, *a* and *c*, being considered for merging is really the merge of two or more rows, then the distance between *a* and *c* is calculated as the distance from *c* to the lowest point which any member of *a* has occupied in the original layout.

To implement this algorithm two subroutines are used, merge and forbid. Merge is the main algorithm; it calls forbid. The one argument to merge determines whether we are merging rows or columns.

Merge proceeds by finding succeeding rows (or columns) from the highest coordinate value to the lowest. Having found two adjacent rows which are within space of one another, it checks for constraint one. If this constraint is met, it checks for the next two constraints by forming a set of "forbidden regions" for each row using forbid. We note at this point that the highest of these two adjacent rows may not, in fact, be a single row, but a set of rows which are considered to have already been merged. In this case all elements in the row are considered to have the same coordinate value.

Forbidden regions are segments along the x-axis (if we are merging rows or the y-axis if we are merging columns) in which no NC or P of the adjacent row (or column) to be merged may lie, if

constraints two and three are to be met. Thus, once forbidden regions are formed for a row, we need only check that one coordinate for each member of the adjacent row does not lie in a forbidden region.

Forbidden regions for the highest of the two adjacent rows are formed when the cases shown in figure 2-56 arise. Brackets indicate forbidden regions. For the next row they are produced for



Figure 2-56

the cases shown in figures 2-57. Thus the rows in figure 2-58 would not merge, and constraints two and three are met.

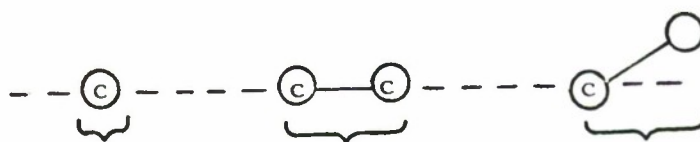


Figure 2-57

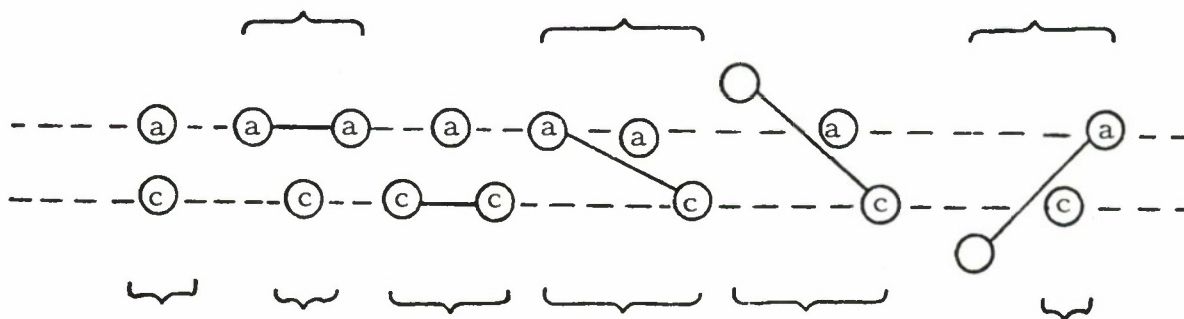


Figure 2-58

When two adjacent rows are found to be compatible, they are merged. Merge puts together as many rows as possible before calculating the resultant row (or column) coordinate for this set. The resultant coordinate is either the coordinate of the NC occurring in this set (by constraint one, there is only one), or, if no NC exists for the set, the resultant coordinate is the midpoint between the highest and lowest row (or column) coordinates in the set.

After experimentation with this algorithm, the question has arisen as to the effect of the directional processing of rows (and columns) it includes. The algorithm always proceeds top to bottom (or right to left) and, quite clearly, the result is somewhat dependent on this direction. It might therefore be worthwhile to try a reversal of directions (bottom to top, for example), or, to include two passes, one in each direction. The results might prove better than those given by the present implementation.

To conclude this section, we give several figures illustrating the use of the MOD system. Figure 2-59 depicts a MOD Input main frame and defines frame in use. Figure 2-60 shows a MOD Output frame in use. And figures 2-61 and 2-62 show sequences of layouts produced by MOD Input and Output using the pretty and merge operations.

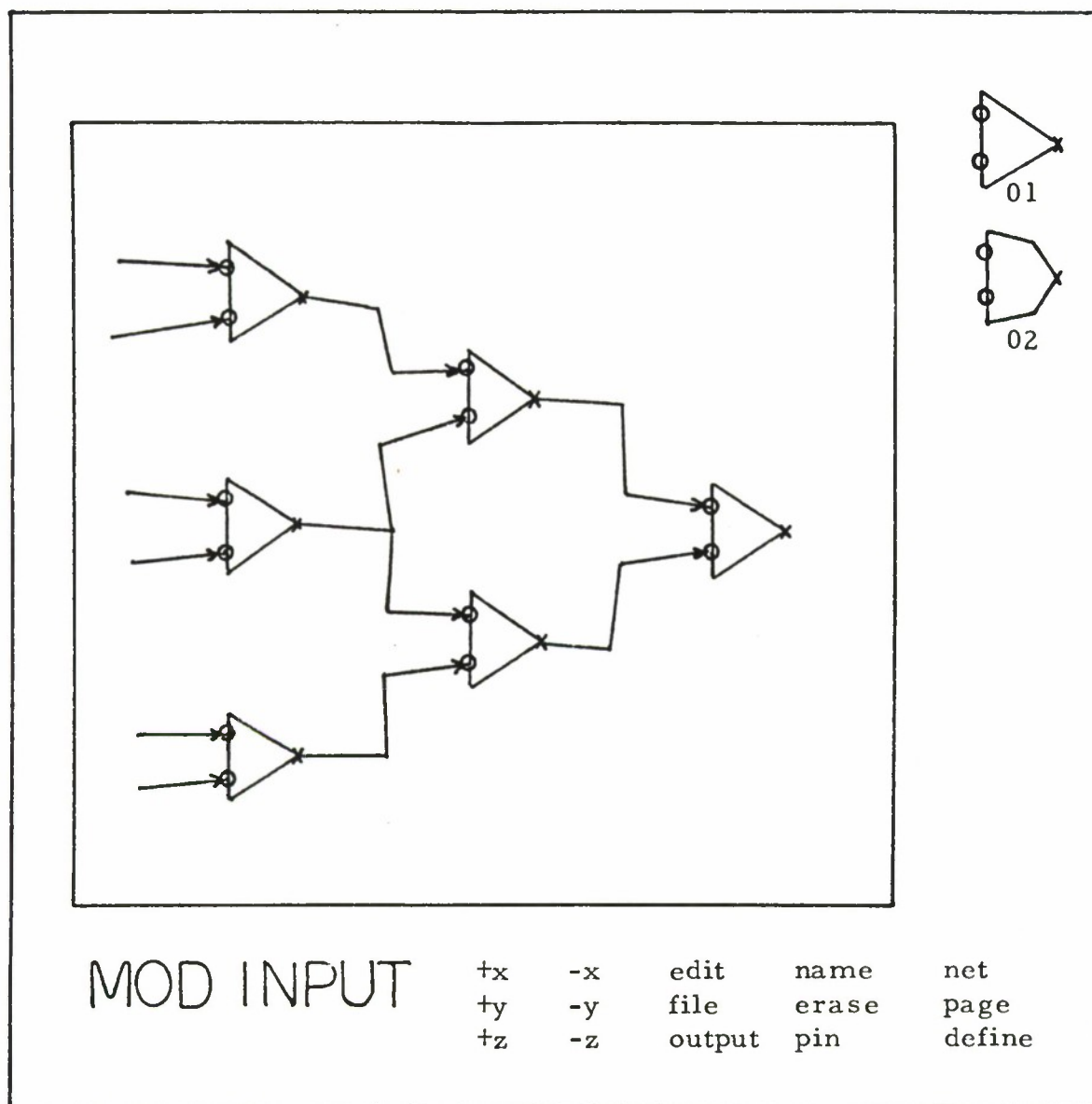
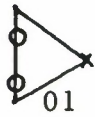


Figure 2-59

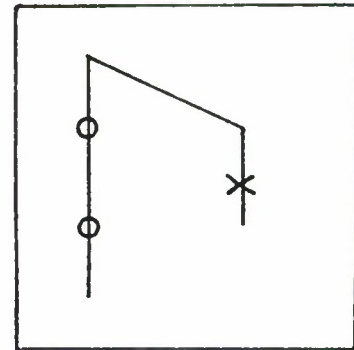


01



02

LINE
FUNCTION
ADP
EP



DEFINED SHAPES

INPUT
COPY
REMOVE
FILE
EDIT

Figure 2-59 (continued)

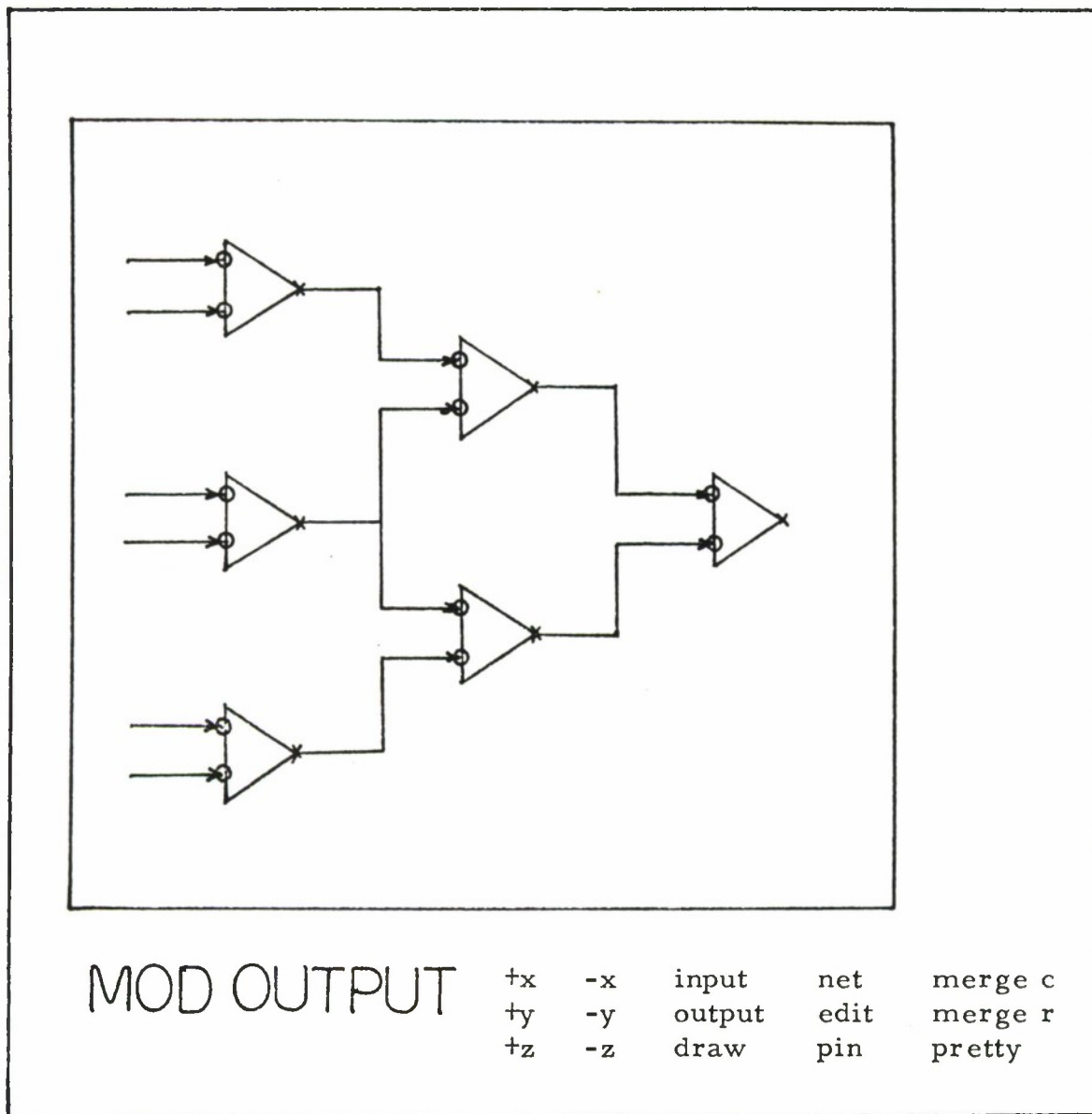
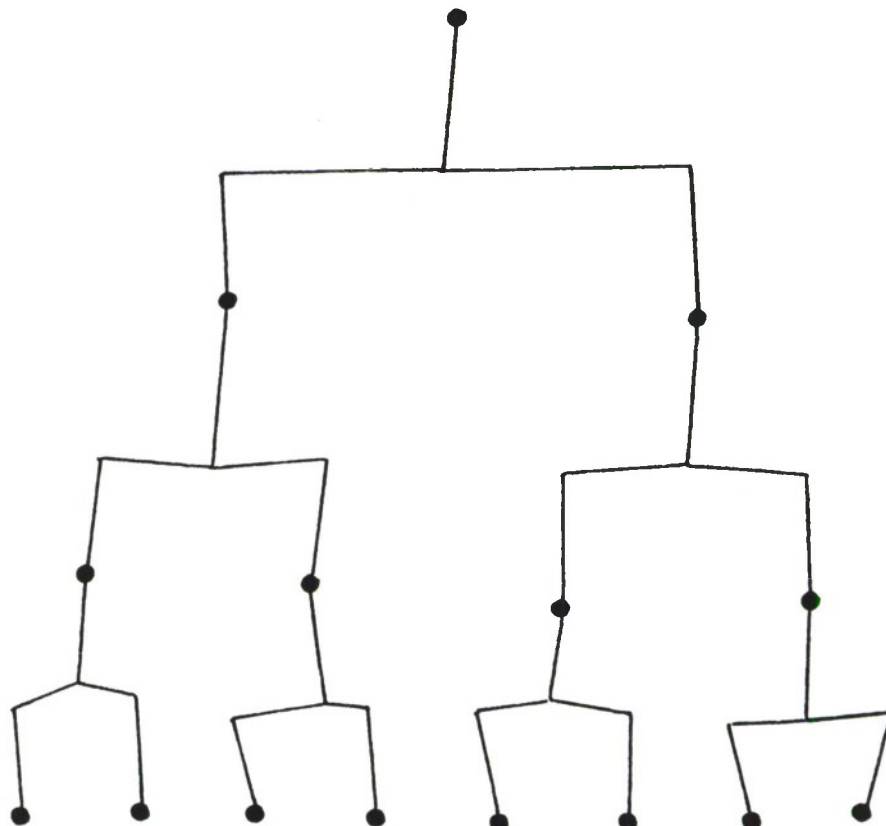


Figure 2-60

input:



pretty:

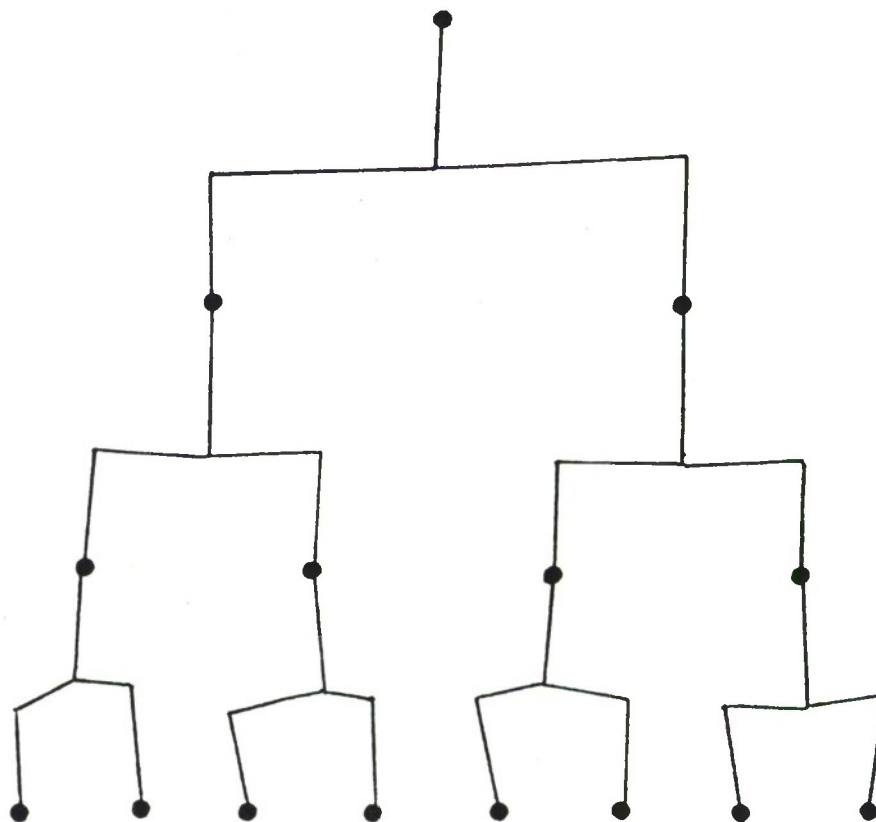
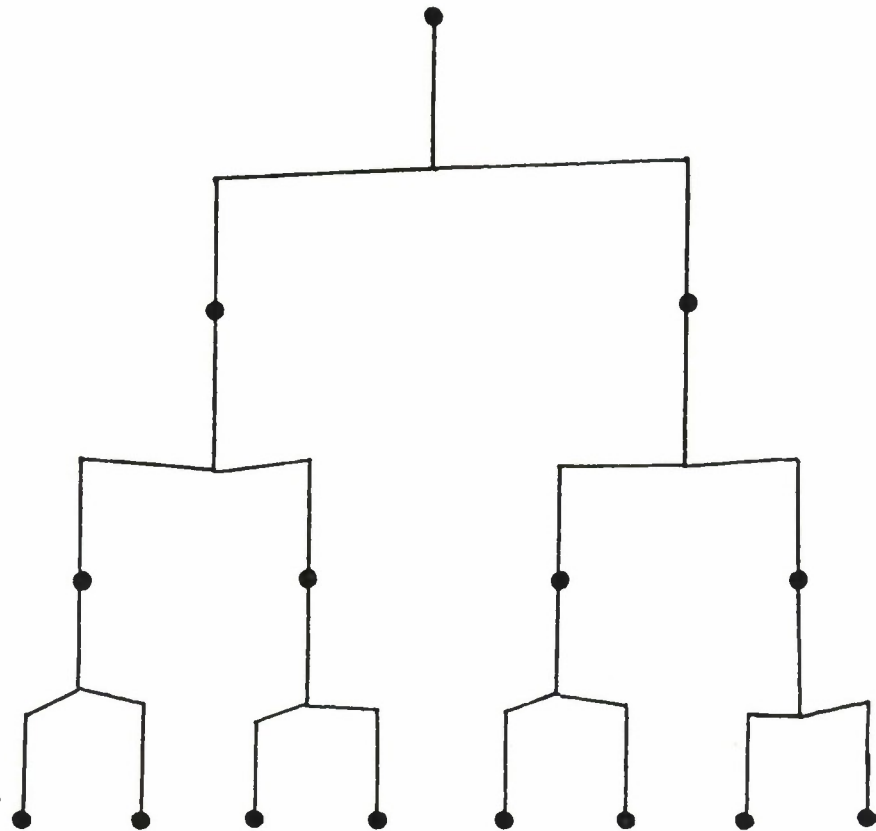


Figure 2-61

merge c:



merge r:

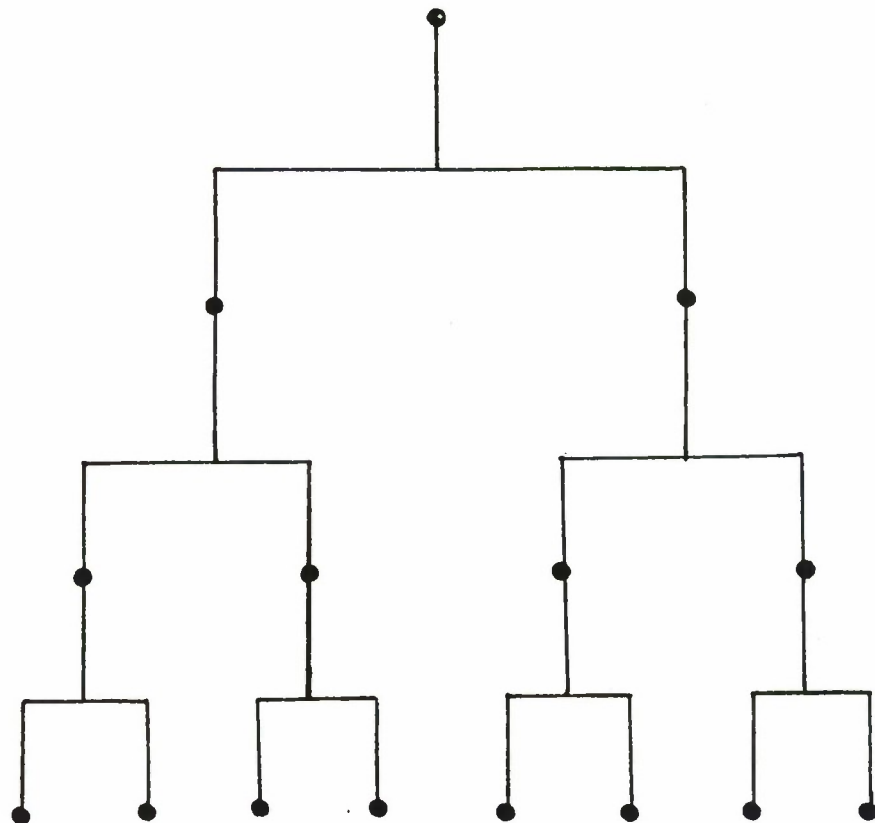
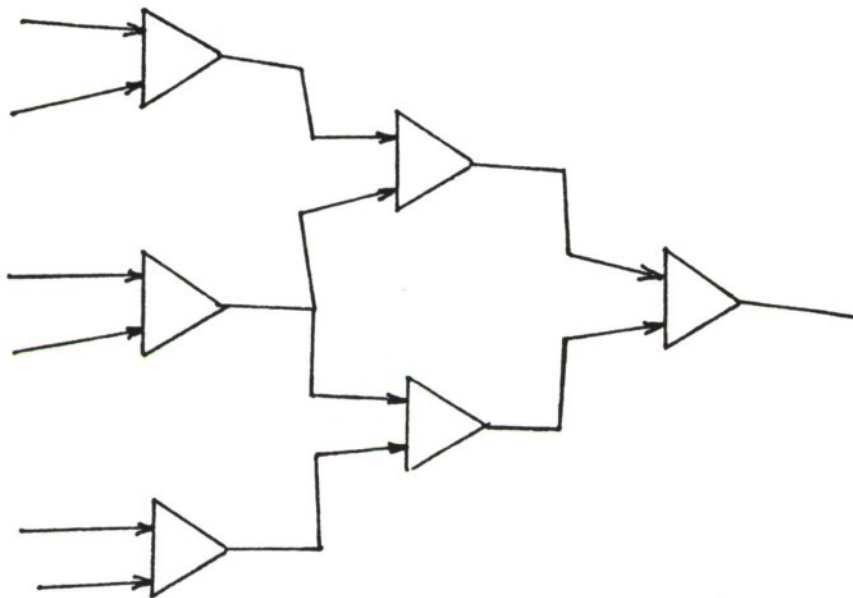


Figure 2-61 (continued)

input:



pretty:

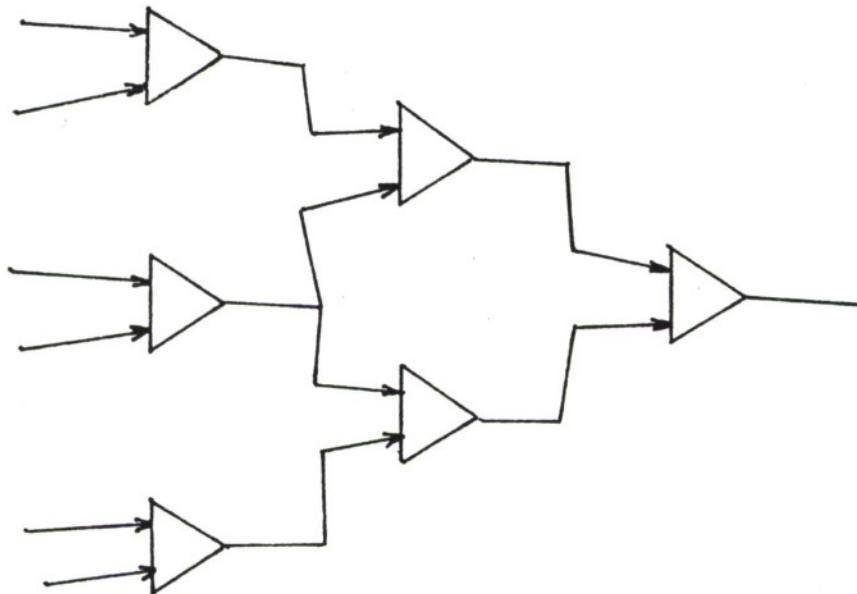
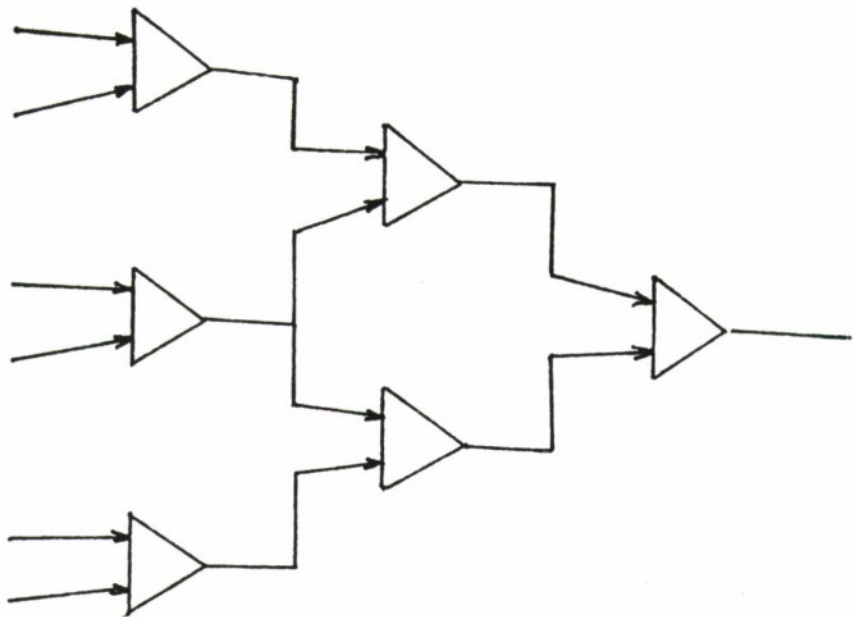


Figure 2-62

merge c:



merge r:

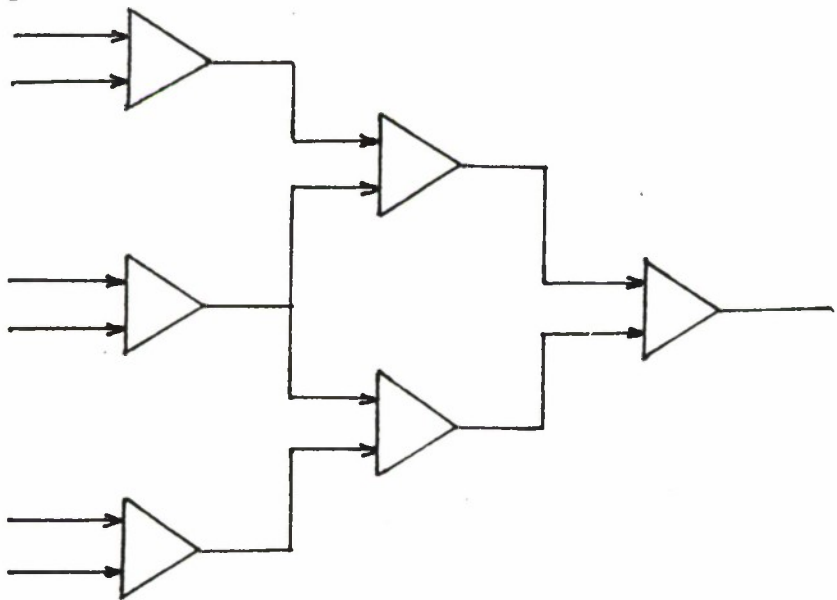


Figure 2-62 (continued)

Chapter 3

APPLICATION DEPENDENT LAYOUT

The problem of layout generation and modification for graphs may be approached from a viewpoint totally different from that used in chapter 2. This new approach considers the problem from an application dependent point of view. By this we mean that we will examine the problem of layout for specific layout types which are dependent, to some degree, on the application in which these layouts are used.

Section 3.1 discusses the advantages of using such an approach. We find that the possibility exists with this approach, for some layout types, of developing algorithms which are in some sense more effective than those developed in chapter 2. Section 3.2 discusses actual layout types and methods to classify layouts. Here we attempt to examine criteria and algorithms for the layout of specific layout types. For several layout types work has already been done, and this work will be mentioned. Finally, section 3.3 briefly considers how a system like MOD may be modified to take advantage of the knowledge of layout types in laying out graphs.

3.1 APPLICATION DEPENDENCY - A JUSTIFICATION

When we examine the value of a study of layout methods, we find that we must consider some of the actual applications in which layout algorithms for graphs might be used. For example, we might consider a graph layout as a circuit layout, flow chart, organization chart, or AMBIT/G data. There are several reasons for this application dependent view of layouts, besides the practical value of the algorithms which might result.

Foremost is the fact that each application carries with it certain conventions about what a layout should look like. For example, how a layout is oriented or how links are drawn may be part of the convention. The conventions of an application may also tell us which of the layout qualities discussed in section 2.1 are to be given priority, and which qualities are to be ignored. In circuit layouts, for example, the qualities most often considered are total link length and the number of intersections; other qualities are relatively unimportant.

Not only may the layout quality priorities be dictated by the particular application, but specific layout characteristics may also be given by these conventions. As seen in the design of the MOD system, in order that it be as general as possible, several different layout characteristics were allowed in the system. In a particular application, however, we are dealing either with, say, undirected or

directed links, nodes with or without shape, and so on. In other words, given a specific application, we need only concern ourselves with layouts containing particular characteristics.

Furthermore, we may know, for a given application, the graph theoretic complexity of the graphs underlying the layouts used. By this we mean the type of graph theoretic structure of the graphs underlying the layouts used in a given application. For example, organizational charts tend to be based on trees or graphs without cycles. But the notion of underlying graph theoretic complexity must be modified when we talk about layouts. We must really consider, instead, what we shall call "apparent underlying graph theoretic complexity" in the layouts of a given application. What we mean here is the amount of underlying graph theoretic complexity which is relevant to the layouts of a given application. For, although the underlying graph of layouts for an application may be quite complex, this complexity may be totally ignored in the layouts. Flowcharts are an example of this phenomenon, for, although the underlying graphs for flowcharts may be quite complex, flowcharts are, in general, arranged so that the nodes are placed in a linear fashion. Thus flowcharts have little apparent graph theoretic complexity.

The point of this discussion is that with any application for which we know layout quality priorities, layout characteristics, and apparent underlying graph complexity, we certainly have a greater

potential for producing more effective layout algorithms than we do by treating the most general case, as we have done in chapter 2. Certainly, we may better tailor the various algorithms to these particular priorities, characteristics, and complexities. Furthermore, we find that many of the considerations which led us to look for modifying algorithms rather than constructive algorithms in section 2.2.2 are no longer relevant in an application dependent framework. For example, we now may have a knowledge of layout priorities, and perhaps even some means for deciding on layout when an arbitrary choice might otherwise be made. Thus, in an application framework, we may more easily consider the possibility of a single constructive layout algorithm (as defined in section 2.2.2) for each application, rather than depending on an interactive environment in which several modifying algorithms must be applied. In fact, it will be seen that this constructive approach (as well as the semi-constructive approach described in section 2.2.2.7) has been used often in the development of layout algorithms for a few of the particular applications mentioned.

Thus, we find that there may be an advantage in examining layout from an application dependent point of view, in that, within this framework, constructive layout algorithms may be more feasible, and more effective results may be possible from application tailored algorithms.

3.2 APPLICATION DEPENDENT LAYOUT TYPES

In this section our aim is to explore the possibility of producing effective constructive layout algorithms given application dependent information. Before doing this, however, we will consider a method of classification of layout types found in various applications, based on layout characteristics, apparent underlying graph complexity, and layout quality priorities. Examples of layouts used in several fields will then be discussed along with their types. It is within this framework of layout type that we will consider constructive algorithms for application dependent layout. We will see that constructive algorithms are not always feasible. Both new algorithms and previously developed algorithms will be mentioned in this discussion.

3.2.1 Classification of Layout Types

We must first consider the motivation for classifying layouts found in various applications into types. The motivation is twofold. First, we would like some method for comparison of layout algorithms for layouts found in different applications. By classifying layouts according to type, we hopefully will see some kind of gradation in algorithms type corresponding to layout type. This would be a helpful environment in which to better understand what factors add to the complexity of layout algorithms, and, perhaps, how they can be simplified. Secondly, once we have developed the framework for

making such correlations, it may prove to be very useful in the development of new algorithms. For, given a new layout type, we may gain some insight into algorithms for its layout by examining algorithms for layouts of similar types.

The classification itself will be based first on apparent underlying graph theoretic complexity, and then on layout characteristics and quality priorities. For example, the main typing "tree," meaning layouts whose underlying graphs are treated as graph theoretic trees, might be divided into subtypes based on particular layout characteristics and priorities. We might then develop several specific algorithms, one for the layout of each different subtype, each of which might have some basic process in common with the others, due to the fact that all are classified under the general typing "tree."

We would then like some ordering of these layout types, hopefully, such that some gradation can be observed in the corresponding layout algorithms. Let us try ordering the types according to apparent underlying graph theoretic complexity, and look for a gradation in corresponding layout algorithms. Those layouts with simpler apparent underlying graphs, are referred to as simple layout types, and those with more complex apparent underlying graphs will be known as complex layout types.

Before looking for this correlation, however, we must decide what algorithm characteristics we should look for a gradation in.

We want algorithm characteristics which reflect the complexity of the algorithms to some extent, although what we mean by algorithm complexity is not quite clear. However, two algorithm characteristics might be appropriate, namely, the amounts of what we shall call structural rigidity and constraint optimization.

The "amount of structural rigidity" in an algorithm means the extent to which algorithm predetermines the layout. For example, at one extreme we have layout types whose algorithms assign the nodes a linear placement, no matter what the input graph is, for example, a flowcharting algorithm. Such an algorithm will be called structurally rigid. On the other hand, we might have layout types whose algorithms have no predetermined mold which layouts follow, for example, those most often used for circuit layouts. Such algorithms have little structural rigidity. Structural rigidity effects the complexity of a layout algorithm, in that, if there is large amount of structural rigidity, the algorithm will probably proceed in a more straightforward manner to obtain a layout, than if there is little structural rigidity, since the layout will be less dependent upon the input, in general.

The meaning of the "amount of constraint optimization" in an algorithm is more obvious. For some layouts little constraint optimization is necessary, whereas for others, layout algorithms may be totally based on constraint optimization. The amount of constraint

optimization in an algorithm is also a good reflection of algorithm complexity, for, in general, the more constraint optimization present in an algorithm, the more complex it will be.

Structural rigidity in an algorithm seems to be at the opposite pole to constraint optimization, for, while the former is based on the assumption that a certain result can always be obtained no matter what the input, the results from the latter are totally dependent on the input, and no amount of success is guaranteed.

In fact, a correlation appears between these two algorithm characteristics. For, where a structurally rigid algorithm may be used, there is little constraint optimization required; and, where much constraint optimization is required, there can be little structural rigidity. Thus we may combine these two factors into one scale for measuring algorithm complexity.

Now, if we consider our original plan for ordering layout types, that of ordering by apparent underlying graph theoretic complexity, we find the following correlation with algorithm complexity (as measured on the structural rigidity-constraint optimization scale). In general, the simpler the layout type in terms of apparent underlying graph theoretic complexity, the greater the possibility for structural rigidity in layout algorithms, and the less the need for constraint optimization in order to obtain layouts. This is clear since, the simpler the apparent underlying graph theoretic complexity

of a layout type, the more we can predict about the structure of the apparent underlying graph (the structure we must consider in layout), and thus, the more we can predetermine what manipulations are necessary for layout and what final forms layouts may take. For more complex layout types, less of the structure of the apparent underlying graph may be predicted, and thus, layout algorithms must depend more on testing and constraint optimization. Thus with the simplest layout types, a layout algorithm may be a simple, rigid, and direct procedure for layout, whereas, with the most complex layout types, layout algorithms may consist of complex optimization procedures.

With this correlation and ordering in mind, we must then consider how to rank the apparent underlying graph theoretic simplicity of layouts in different applications. There are several factors which might be considered. For example, we might take into account whether the apparent underlying graphs have cycles or not, whether they are planar or not, whether they are separable (i. e., can be separated into components as in the algorithm of Appendix 3) or not, and so on.* The ordering will be as follows. Near the bottom of the

* Again, by apparent underlying graph, we mean the structure of the underlying graph which is relevant in layout. So that, for example, if we do not care in a layout process whether the underlying graph has cycles or not, then the apparent underlying graph is considered not to have cycles.

scale we will consider the simplest layout types, those whose apparent underlying graphs are without cycles (and which are therefore planar and separable). We will name this layout type the tree type after the graph-theoretic name of the structure of its apparent underlying graph. We will see that algorithms for tree layout are quite straightforward and structurally rigid. At the top of the scale, we have layout types with apparent underlying graphs which may be non-planar, may contain cycles, and may be non-separable. The most complex type are layouts with such apparent underlying graphs and which use both ADP's and EP's (as defined in section 2.3.1), thus requiring that links be placed around a node in a certain order. We expect that constructive layout algorithms for such complex layout types will consist mainly of constraint optimization, and, will be very difficult to develop, if not impossible.

We will not consider a complete categorization of all possible layout types, as such a task seems infeasible. We will attempt, however, to order several common layout types, according to their apparent underlying graph theoretic complexity and thus according to their respective layout algorithm complexity. The intention here is only to experiment with the possibility of a layout classification, rather than to strictly define it.

3.2.2 Layout Types and Algorithms

What follows is a description and classification of various commonly used layout types, along with a discussion of algorithms for the layout of these types. The layout types are described in order of layout type complexity. A corresponding gradation in layout algorithm complexity is also seen. Five layout types are presented.* We start with the simplest types and proceed to the most complex. With each type several subtypes may be included.

3.2.2.1 Linear Layouts

The linear type is the layout type in which there is the least apparent underlying graph theoretic complexity. In other words, the apparent underlying graph consists of a simple string of nodes, perhaps connected by links, or, it is treated as if it does. The only general constraint of this layout type is that the nodes be placed in some linear arrangement. Thus the basic form of a linear layout is quite rigidly predetermined.

As mentioned above, flowcharts are an example of this layout type. By convention, nodes of flowcharts generally assume a linear placement, regardless of underlying graph theoretic structure. Often

*The type names given to the various layout types, in general, have no relationship to definitions commonly found in graph theory. When there is a correspondence with a type name and a graph theoretic definition, as in the case of "tree layouts," it is pointed out in the text.

nodes are aligned in a vertical arrangement. One complication in flowchart layout, however, is that of link routing. In general, links are required to be drawn as series of horizontal and vertical segments. This may not always be possible.

The usual requirements of an algorithm for flowchart layout, then, are that, given the order in which nodes are to be placed, the algorithm must decide on a linear placement, and then must route links along horizontal and vertical runs, where possible. Often, where the origin and destination of a link are far apart, intermediate reference nodes are allowed. These are nodes especially created and labeled in pairs. They imply that a link which ends at such a node really continues with the link which originates at the other reference node with the same label. Link intersections are generally allowed, and thus, the only difficulty of such an algorithm is in checking that no link segments are drawn on top of others previously drawn (in some cases the horizontal-vertical requirement may have to be broken to avoid this). Many such algorithms have been written and implemented as computer packages for flowchart generation, for example, System/360 Flowchart by IBM (19).

In this discussion we have been careful to separate out the task of deciding on linear node order, from that of generating a layout given an order. It is felt that the manner in which node order is decided upon is not relevant to the actual problem of layout, and

seems to depend upon the requirements of the particular language for which a flowcharting program is to be used and on the particular priorities of the user.

It is clear, then, that for this example of the linear layout type, straightforward structurally rigid constructive algorithms without optimization are easy to design. Although algorithms may include optimization procedures for minimizing the number of non-horizontal and vertical link segments, such optimization is not necessary for flowchart layout. An example of a typical flowchart is shown in figure 3-1.

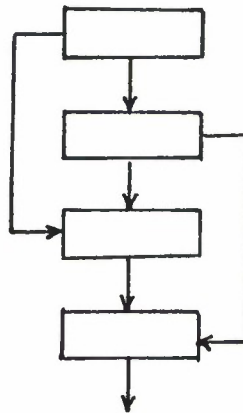


Figure 3-1

3.2.2.2 Tree Layouts

The tree type includes layouts which are more two-dimensional than the linear layouts, but the underlying graphs for this type may contain no cycles. In forming tree layouts, underlying graph-theoretic structure is relevant in determining node position. If a graph is a

tree in the graph-theoretic sense, we are guaranteed that it may have a planar representation (no intersections) and may be laid out in one of many predetermined, somewhat rigid forms without complication. This will be demonstrated below.

In most applications tree layouts are formed in levels as follows. Some single node is designated as the "root" node. The root node comprises the first level. Each node adjacent (in the graph theoretic sense) to the root lies in a level higher than the root node level. They may or may not all lie in the next level, but at least one does. In turn, for each node, say *b*, adjacent to a node, say *a*, which is closer to the root node than *b* in terms of graph theoretic distance, node *b* lies at a higher level than node *a*.^{*} It is guaranteed that there will be no level conflict because there are no cycles; hence, nodes may be partially ordered with respect to any root node.^{**}

Nodes without successors in this description are known as leaves.

Algorithms for general tree layout must, given a root, separate the nodes into levels and place them accordingly. In general, levels are spaced at regular intervals.

Tree layouts are used in many applications to represent

^{*}Node *a* is then called the "father" of node *b*, and node *b* is called the "son" of node *a*.

^{**}Tree layout does not require that the nodes be totally ordered.

hierarchical structure or dependency relationships. Each application may carry with it additional requirements as to what tree layouts should look like. Additional requirements regarding horizontal and vertical placement of nodes are often found. For example, one application may require that a node lie in the level immediately next to its father node (leveled trees), or conversely, that a node lie in the highest level possible, the level just before the lowest of any of its sons, and that all leaf nodes lie in the same level (unleveled trees). Or, we may have the requirement that a node must be centered over its sons (son-centered), or, on the other hand, that nodes in a given level and links passing through the level be placed evenly spaced in the level around some center, regardless of the placement of sons (level-centered). Furthermore, bends in links may or may not be allowed.

For example, in linguistic applications, a phrase marker representing the parse of a sentence is often depicted by an unleveled, son-centered tree layout without bends, as shown in figure 3-2a. Whereas, tree layouts used as organization charts are often leveled, level-centered trees (with bends allowed, although no bends will appear) as shown in figure 3-2b. It is of interest to note, however, that with tree layouts, as with linear layouts, layout characteristics as defined in section 3.1, such as directedness of links, are of little importance in the layout process.

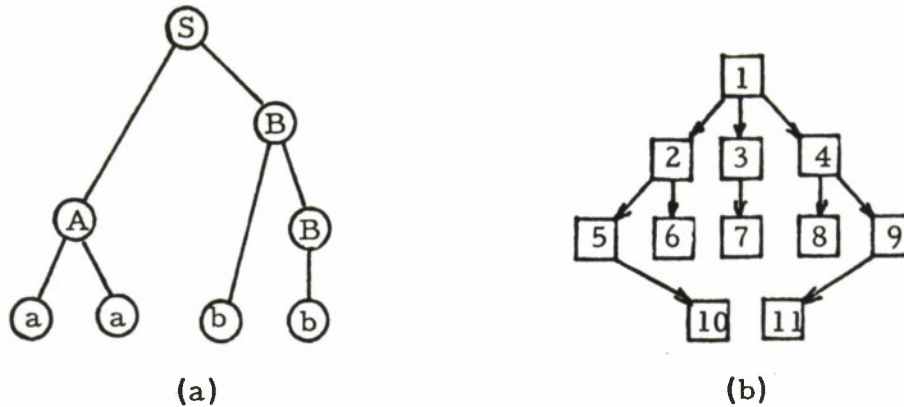


Figure 3-2

Again, for clarity, we will repeat the definitions of these layout requirements:

- 1) Leveled trees: every node lies in the level immediately next to its father node.
- 2) Unleveled trees: every node lies in the highest level possible, the level just before the lowest of any of its sons, and all leaf nodes (nodes without sons) lie in the same level.
- 3) Son-centered trees: every node is centered over its sons.
- 4) Level-centered trees: nodes in a given level and links passing through that level are evenly spaced around some center.

Since each of the application dependent requirements mentioned above affects the formation of tree layouts, they must be considered in the formulation of layout algorithms.* Most tree layout

*In the following algorithm description we will talk about tree layouts oriented with roots on top and leaves on the bottom; however, this discussion may also apply to tree layouts with other orientations, such as left to right.

algorithms, however, may consist of two main steps:

- a) place nodes into levels to determine y-coordinate, and
- b) place nodes within levels to determine x-coordinate.

The details of these steps differ according to whether the tree layout is to be leveled or unleveled, son-centered or level-centered, and whether or not bends are allowed. Each such variation is considered as a separate tree subtype.

An algorithm for the layout of all of these tree subtypes will now be presented. The algorithm consists of the two basic steps (a and b) mentioned above. As stated, there are variations in these two steps according to the subtype of tree layout desired. Step a has two variations, a.1 for all leveled layouts, and a.2 for all unleveled layouts. In the following description of these variations of step a, the symbol "# " is used as a place marker.

Step a.1: for all leveled subtypes, place the node in the lowest level possible as follows: form ordered list of nodes, S_i 's, $i = 0, \dots, n$, one for each level, as:

i) $S_0 = (\text{root node})$

ii) If $S_i = (K_1, K_2, \dots, K_m)$ and for some $K_i, K_i \neq \#$,

then $S_{i+1} = (K_{1_1}, K_{1_2}, \dots, K_{1_{n_1}}, K_{2_1}, \dots, K_{m_{n_m}})$ where:

$$K_{j_k} = \begin{cases} \#, & \text{if } K_j \text{ is } \# \text{ or if } K_j \text{ has no sons} \\ \text{the } k\text{th son of } K_j, & \text{if } K_j \text{ has any sons} \end{cases}$$

$$n_j = \begin{cases} 1, & \text{if } K_j \text{ is } \# \text{ or if } K_j \text{ has no sons} \\ \text{the number of sons of } K_j, & \text{otherwise} \end{cases}$$

If, for all K_j in S_i , $K_j = \#$, then $n = i - 1$.

Step a.2: for all unlevelled subtypes, place the nodes in the highest level possible, as follows: form ordered lists of nodes, S_i 's, $i = 0, \dots, n$, one for each level, as:

i) $S_0 = (\text{root node})$

ii) If $S_i = (K_1, K_2, \dots, K_m)$ and some K_j has a son, then

$S_{i+1} = (K_{1_1}, K_{1_2}, \dots, K_{1_{n_1}}, K_{2_1}, \dots, K_{m_{n_m}})$ where:

$$K_{j_k} = \begin{cases} K_j, & \text{if } K_j \text{ has no sons; } K_j \text{ in } S_i \text{ is then} \\ \text{changed to a } \# & \\ \text{the } k\text{th son of } K_j, & \text{if } K_j \text{ has any sons} \end{cases}$$

$$n_j = \begin{cases} 1, & \text{if } K_j \text{ has no sons} \\ \text{the number of sons of } K_j, & \text{otherwise} \end{cases}$$

If, for all K_j in S_i , K_j has no sons, then $n = i$.

Variations in step b then complete the algorithm description.

There are six variations, one for each possible combination of the

layout requirements mentioned above.* These six variations are given below as steps b.1 through b.6. In the description of these steps, c is some x -coordinate arbitrarily chosen as a fixed center for the layout. Once the K_j 's are positioned in step b, only the K_j 's such that $K_j \neq \#$ actually appear in the layout; the others simply act as guides for links, or are ignored.

Step b.1: for all leveled, son centered subtypes:

i) Evenly space** all the K_j 's in S_n around the fixed center c .

ii) For each S_i in the order $i = n-1, n-2, \dots, 0$, place each node $K_j \neq \#$ in the center of the x -coordinates of the elements which it generated in $S_{i+1}, K_{j_1}, \dots, K_{j_{n_j}}$.

Step b.2: for all leveled, level-centered subtypes: for each S_i place the K_j 's $\neq \#$ evenly spaced and centered with respect to the fixed center c .

In both steps b.1 and b.2, links are then drawn between each of the pairs $K_j \in S_i$ and $K_{j_k} \in S_{i+1}$, where $K_{j_k} \neq \#$, for $i = 0, \dots, n-1$.

* In the leveled subtypes no bends can occur since links travel only between adjacent levels. Hence we have six combinations of the requirements, instead of eight, as would be expected.

** The details of this spacing operation will be worked out when the algorithm is implemented, and will not be bothered with here.

Step b.3: for the unlevelled, son centered subtype with bends:

- i) Evenly space all the K_j 's in S_n around the center c .
- ii) For each S_i in the order $i = n-1, n-2, \dots, 0$ place each K_j in the center of the x-coordinates of the elements which it generated in S_{i+1} .

Step b.4: for the unlevelled, level-centered subtype with bends:

for each S_i , place all the K_j 's evenly spaced and centered with respect to the fixed center c .

In both steps b.3 and b.4, links are then drawn between each of the pairs $K_j \in S_i$ and $K_{j_k} \in S_{i+1}$, $i = 0, \dots, n-1$.

Step b.5: for the unlevelled, son-centered subtype without bends: there is a problem with this subtype, in that intersecting lines may be created. If, for example, we represented the underlying graph of figure 3-3a in this manner, we would obtain that of figure 3-3b. To remedy this situation, let us first form the layout as for the

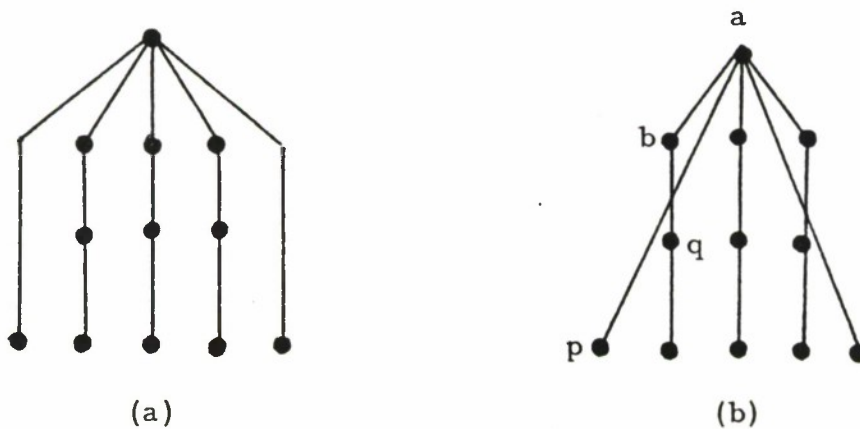


Figure 3-3

unleveled, son-centered subtype with bends, and put in single segment links, father to son as in 3-3b. We may then remove any resulting intersections using the procedure shown in figure 3-4.

The basic idea of this procedure is that if two links, say (a,p) and (b,q) , intersect, as in figure 3-3, where a and b are the node labels of the topmost node of each link, and the level of a precedes that of b , then, by moving up the y -coordinate of the level of a and that of all other levels above a , the intersection may be removed. It should be noted that in order to avoid intersection with this subtype we can no longer guarantee that levels will be evenly spaced. Hence, we are sacrificing some structural rigidity.

Let us discuss the intersection removal procedure in more detail. First, notice that if any two links, say again (a,p) and (b,q) , intersect in the unleveled, son-centered subtype without bends, it must be that either a is an ancestor^{*} of b , or that b is an ancestor of a . Proof of this fact is sketched as follows:

Lemma: Let links (a,p) and (b,q) intersect in the unleveled, son-centered subtype without bends, as described above. Then either a is an ancestor of b , or b is an ancestor of a .

Proof: Suppose that (a,p) and (b,q) intersect, where a and b are the lower level nodes in each of these links. We will

^{*}The term ancestor is used to mean father, or grandfather, or great-grandfather, and so on.

assume that a is not an ancestor of b , and that b is not an ancestor of a , and show that this leads to a contradiction.

First note that for every node z not in the highest level (level n), by our formulation of node order in levels (step a for the unleveled subtypes), all leaves of node z are in sequential order in level n . Let us then define a range $R_n(z)$ for node z in level n , as the range along the x -axis in which leaves of node z lie. Note that, by our formulation, only nodes which are progeny^{*} of node z lie in $R_n(z)$. Now, since nodes are son-centered in this subtype, all progeny of z lie in the area, $A_n(z)$, over $R_n(z)$ running from the level of z to level n . Thus all links from z to nodes in higher levels lie within $A_n(z)$.

Furthermore, only nodes which are progeny of z lie in $A_n(z)$. For, assume that this is not the case and that some node w lies in $A_n(z)$ and is not progeny of z . According to our formulation of $R_n(z)$, w is not in level n . Furthermore, due to the son-centered requirement, it must be that $R_n(z) \cap R_n(w) \neq \emptyset$. Then there must be a node v in level n such that $v \in R_n(z) \cap R_n(w)$ and such

* Again, progeny is used to mean sons, grandsons, great-grandsons, and so on.

that v is progeny of both z and w . Now, since each node has only one father, we may trace from node v back through its forefathers in sequence starting with v 's father. If node w appears first in this sequence, then v cannot also be progeny of node z , since w isn't. If node z appears first in this sequence, then by formulation of levels, z must be progeny of w , and $w \notin A_n(z)$. Hence we have a contradiction, and $w \notin A_n(z)$, if w isn't progeny of z .

Now, if a is not an ancestor of b , and b is not an ancestor of a , then $b \notin A_n(a)$ and $a \notin A_n(b)$. Hence the two areas, $A_n(a)$ and $A_n(b)$ are distinct, and no link from a to a higher level can intersect with any link from b to a higher level.

QED

Thus it must be that either a is the ancestor of b (denoted by $a = f^*(b)$) or vice versa. Suppose, then that $a = f^*(b)$. Then, certainly, a and b could not be at the same level, and, in fact, the level of a (denoted by $L(a)$) must be smaller than that of b .

We have then determined that for two links, (a, p) and (b, q) , to intersect, $L(a) \neq L(b)$, and either $a = f^*(b)$ or $b = f^*(a)$, depending on whether $L(a) < L(b)$ or $L(b) < L(a)$, respectively. The method for intersection elimination in figure 3-4 uncovers intersections in a

layout by looking for b's starting with level $n-1$ and proceeding to lower levels, where the b's are assumed to be the lower of the topmost nodes, a and b, of intersecting links. While considering each node as a b, all of the node's ancestors, starting with immediate and proceeding to more distant, are considered as a's. We are guaranteed that if, say, $L(a) < L(b)$, then we can eliminate the intersection if we move $L(a)$ up far enough, along with all levels above it. The idea is that we move $L(a)$ up far enough so that the link (a, p) clears $A_n(b)$. This can always be done, since p cannot lie in $A_n(b)$ (otherwise p would have two fathers). Thus if we make the link (a, p) vertical enough (by moving a up and leaving p fixed, where, of course, $L(p) > L(b)$) it must clear $A_n(b)$.

The only problem left to worry about with this procedure is that no new intersections are created by such a move. In fact, we find that new intersections may be created by such a move, but that they are such that the topmost node involved in the new intersection, a' , must be such that $L(a') \leq L(a)$, since no nodes in levels greater than $L(a)$ are effected by the move. Even better, we find that $L(a') < L(a)$, since, if we are moving $L(a)$ up, the move has the same effect on all intersections with topmost node in $L(a)$ as it does on the intersection involving a . Thus, if we process the layout for intersections, bottom up, we may remove them all in one pass.

In the algorithm, then, for each (a, p) and b considered, the process checks the cases listed below to see if (a, p) intersects with some link downward from b . The line $y = \lambda(L(b))$ is the horizontal at the level of b , and $y = \lambda(n)$ is the horizontal at level n . The line $x = x_1$ is the vertical at the left of the range $R_n(b)^*$, and $x = x_2$ is the vertical at the right of $R_n(b)$. The cases are:

a) If $x_1 = x_2$, then there is an intersection if and only if (a, p) intersects the line $x = x_1$ between $y = \lambda(L(b))$ and $y = \lambda(n)$.

b) If $x_1 \neq x_2$, then there is an intersection if and only if either:

1) All of the following three conditions hold:

i) (a, p) intersects $y = \lambda(L(b))$ between $x = x_1$ and $x = x_2$,

ii) (a, p) intersects either $x = x_1$ or $x = x_2$ between $y = \lambda(L(b))$ and $y = \lambda(n)$, and

iii) (a, p) intersects some horizontal $y = \lambda(k)$, where $L(b) < k < n$, between $x = x'_1$ and $x = x'_2$, where $x = x'_1$ is the vertical on the left side of the range of progeny of b at level k , and $x = x'_2$ is the vertical on the right side of this range.

or
2) (a, p) intersects both $x = x_1$ and $x = x_2$ between $y = \lambda(L(b))$ and $y = \lambda(n)$.

* See definition in preceding proof.

In checking for these cases, the algorithm first decides whether or not $x_1 = x_2$. If $x_1 \neq x_2$, it first checks whether (a,p) intersects $y = \lambda(L(b))$. If it doesn't, case $(b,1)$ is eliminated immediately, and, at that point, the algorithm needs only to check for intersection with one of the sides $x=x_1$ and $x=x_2$ to see if $(b,2)$ holds.

The algorithm shown in figure 3-4 proceeds using the approach in the preceding discussion to remove all intersections from a layout for the unlevelled, son-centered subtype without bends. It processes levels for intersections, starting at the highest (bottommost) level and proceeding to the lowest (topmost). Initially all levels are evenly spaced, but are changed when necessary to avoid intersections. The following notation is used in the flowchart of figure 3-4:

L counts levels looking for b 's

i_L is the i th node in the L th level, left to right

n is the total number of levels

$\lambda(L)$ is the y -coordinate of level L ; the results of the process may be seen in these numbers

YINC is the original distance in y -coordinate between levels

$S(x,y)$ gives the next son of x after son y ; if $y=0$ the first son is given; if $S(x,y)=0$, then y was the last son

$f(x)$ gives the father of x ; if $f(x)=0$ then x is the root

$R_m(x)$ gives two values, x_1 and x_2 , denoting the left and right limits, respectively, of the range along the x -axis which progeny of x occupy on level m ; obviously, if $L(x) > m$ then $R_m(x)$ has no value

Again, we emphasize that this algorithm is applied to a layout which has been formed by proceeding as in the unleveled son-centered subtype with bends, and then removing the bends in the links.

In summary, then, with the unleveled son-centered subtype without bends, we lose some structural rigidity, in that, in order to guarantee that no intersections will occur, we cannot guarantee that levels will be evenly spaced.

Step b.6: for the unleveled, level-centered subtype without bends: there is a problem with intersections in this subtype also; however, here we can find no remedy. Figure 3-5b shows an example of such a layout for the underlying graph of 3-5a. It is clear that moving levels up or down is not a solution. Unfortunately, the method for the unleveled, level-centered subtype with bends may produce

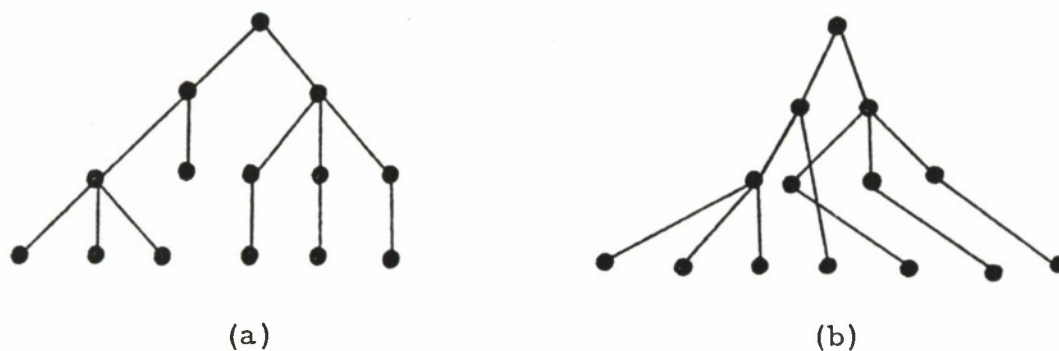


Figure 3-5

more than one bend per link.*

To handle the problem of layout for the bend-free unlevelled, level-centered subtype, we might consider an optimization technique. For example, we might use an algorithm which performs several passes. Each pass may rearrange the order of sons under a father, in order to find an arrangement which yields a layout with the minimal number of intersections possible. Such an approach would not guarantee an intersection-free-layout, but would find the optimal solution, given the constraints of this subtype. Thus we must sacrifice the guarantee of node order in order to meet the layout requirements for this subtype. Even then, we cannot be sure that the requirements will be met.

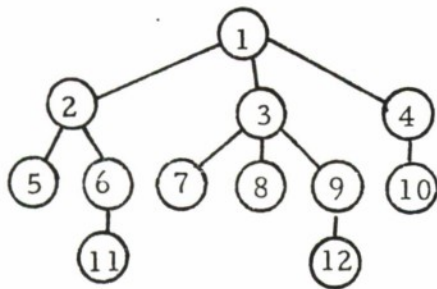
Examples of layout for the leveled subtypes with and without bends and the unlevelled subtypes with bends are shown in figures 3-6a, b, c, and d for the same underlying graph. In this example, the lists, S_i , for the leveled subtypes are:

* We note that we can guarantee at most one bend per link for the unlevelled, son-centered subtype with bends, and, in all leveled subtypes, obviously, no bends occur since links always travel between two adjacent levels. In the unlevelled, son-centered subtypes, it turns out that each $K_j = \#$ is aligned directly above either another $\#$ or the node whose place is marked by this $\#$, due to the son-centered requirement. Thus a straight line may be drawn through the series of $\#$'s over a node making one link segment, and a line may be drawn between the top $\#$ and the father node, completing the link with only two segments.

$$\begin{aligned}
S_0 &= (1) \\
S_1 &= (2, 3, 4) \\
S_2 &= (5, 6, 7, 8, 9, 10) \\
S_3 &= (\#, 11, \#, \#, 12, \#)
\end{aligned}$$

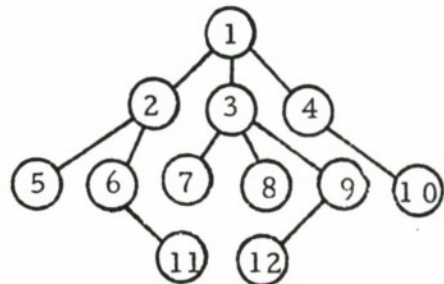
and those for the unleveled subtypes are:

$$\begin{aligned}
S_0 &= (1) \\
S_1 &= (2, 3, \#) \\
S_2 &= (\#, 6, \#, \#, 9, 4) \\
S_3 &= (5, 11, 7, 8, 12, 10)
\end{aligned}$$



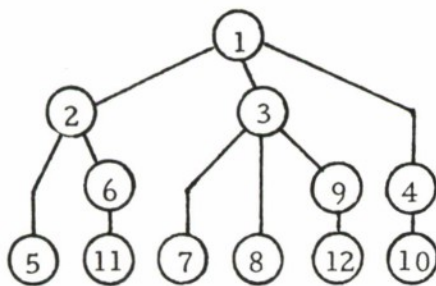
Levelled
son-centered

(a)



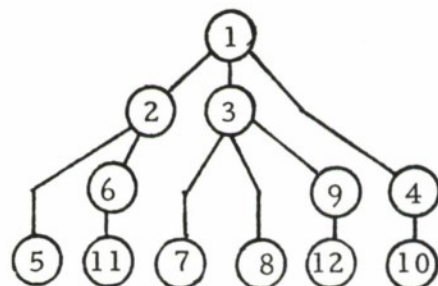
Levelled
level-centered

(b)



Unleveled
son-centered

(c)



Unleveled
level-centered

(d)

Figure 3-6

Figure 3-7 depicts a sequence of layouts generated by the algorithm for removing intersections from the unleveled, son-centered subtype without bends (step b.5). Initially (3-7a) there is one intersection. To remove it, levels 0-2 are moved up (3-7b), causing another intersection. When this is discovered, level 0 is moved up. The result (3-7c) is intersection free.

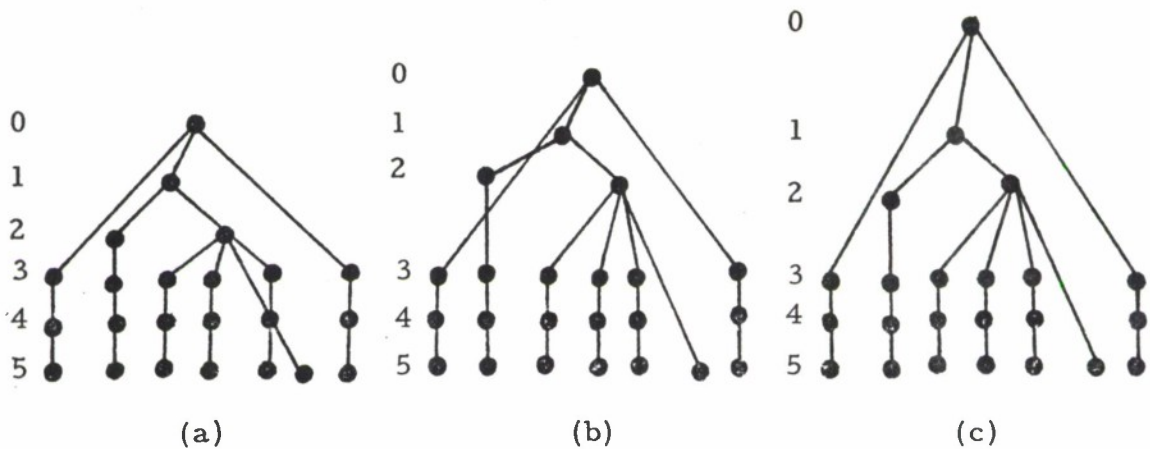


Figure 3-7

We may be assured in all leveled subtypes that no intersections occur. For, since links run only between two adjacent levels, any intersection, say (a,p) and (b,q) must occur between two adjacent levels. Say that $L(a) = L(b) = k$ and $L(p) = L(q) = k+1$. Obviously, it must be that $a \neq b$ (otherwise there would be no intersection). But, by the formulation of S_{k+1} from S_k , if a precedes b in S_k , then the sons of a precede those of b in S_{k+1} , and vice versa. Hence (a,p) and (b,q) cannot cross. In the unleveled subtypes with bends, we have the same assurance, since the # elements

act as nodes in the argument, and links may be considered to travel only between two adjacent levels.

In summary, we have examined several subtypes of the tree layout type and developed constructive algorithms for the layout of each subtype. A gradation was found in complexity of layout according to subtype. With all leveled subtypes, and with the unleveled subtypes with bends, the algorithms are straightforward and structurally rigid. With the unleveled, son-centered, bend-free subtype the layout algorithm is still direct, but the resultant layout structure may not be predicted as rigidly. With the unleveled, level-centered, bend-free subtype, the desired structure cannot always be attained. In fact, with this subtype, a procedure involving optimization, with no guarantee of total success, seems the only feasible approach to meeting the requirements.

3.2.2.3 Network Layouts

The network type includes the set of layouts whose apparent underlying graphs have nodes which are partially ordered by the links which connect them and by the directions of those links. In other words, the apparent underlying graphs are directed. The resultant layouts are to reflect this partial ordering. This implies that, although the undirected structure of the underlying graph may have cycles, the directed graph representing the partial ordering

does not contain directed cycles. This also implies, of course, that the underlying graph may have no self-loops. The underlying graph itself, however, need not necessarily be directed, but information about intended link direction, and hence, the partial ordering of nodes, must be available.

We do not require that network layouts be intersection free, although the number of intersections should be minimized. Nodes, in general, should not have ADP's and EP's which might restrict the order in which links may be placed around a given node. Although, nodes which specify one side for input, and one for output may be used. The resultant layout should optimize linear directional consistency by giving an overall direction to the node placement based on the partial order. It should also minimize the number of intersections which appear in the links between these nodes.

We find then, that in forming network type layouts, some structural rigidity exists in the form of node placement with respect to partial ordering. But some constraint optimization must be used for the attainment of a minimal number of intersections. Layouts generated may include more than one source node (a node at which no arrows terminate), and/or more than one sink node (a node from which no arrows originate).

Network type layouts are common to many applications and are often used to depict logical order or flow, such as in logic

diagrams or PERT flowcharts. Some examples are shown in figure 3-8. In some cases where a network type layout is desired, but the partial node ordering contains directed cycles, as in 3-8c, the directions of some of the arrows are temporarily reversed to remove the cycles as in 3-9a, and once a layout has been achieved, the original link directions are restored, as in 3-9b. In this case, the directional consistency of the result is decreased.

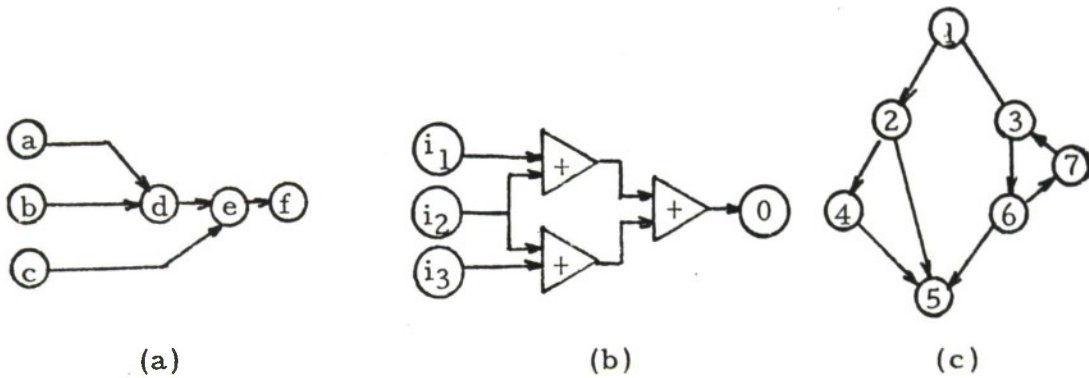


Figure 3-8

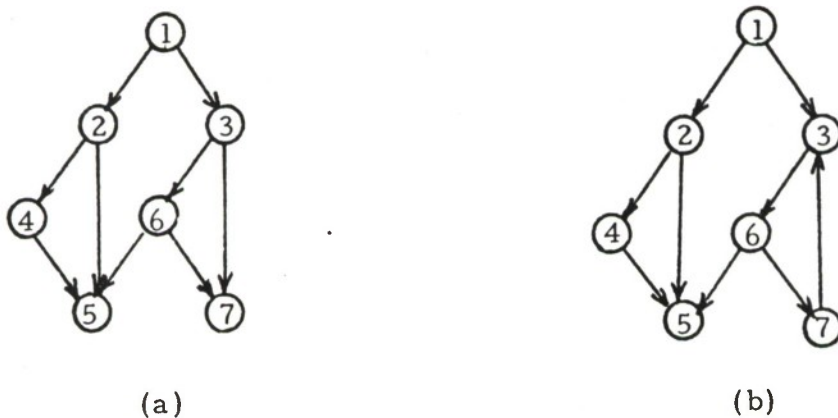


Figure 3-9

By convention, the partial order of the nodes is usually reflected in their placement along either the x- or y-axis. For example, source nodes may be placed to the left, sink nodes to the right, and all others in between, dependent on their place in the partial order. In our discussion of algorithms for network layout, this left to right placement will be used, but algorithms for other orientations may easily be derived from this discussion. In summary, we require that an algorithm for network layout place the nodes in such a manner that x-coordinate reflects the partial ordering. The y-coordinates (and within some limitation, the x-coordinates) may then provide some degree of freedom, so that nodes may be positioned to minimize the number of intersections.

Di Giulio and Tuan (12) have designed a two step algorithm to accomplish these tasks. The first step consists of assigning the nodes to "stages" or groups to be given the same x-coordinate, according to the partial ordering. Stages are numbered so that the lower the number, the smaller the x-coordinate. They are formed as follows:

- 1) Form stage zero, $S(0)$, from the union of all source nodes.

- 2) Form stage n , $S(n)$, from the union of all nodes, y , such that for $x \in S(n-1)$, there is a directed link from x to y in the underlying graph. If the resultant $S(n)$ contains any node z found in a

previous stage, say, $S(n-i)$, $i \geq 1$, mark the occurrence of z in $S(n-i)$ as a "dummy node" (indicated by \boxed{z}), delete all successors of z from stages $S(n-i+1), \dots, S(n)$ unless these nodes are also successors of other non-dummy nodes, and add the dummy node \boxed{z} to each of the stages $S(n-i+1), \dots, S(n-1)$.

3) The process stops when a stage, $S(m)$ is reached for which $S(m) = \emptyset$.

Relative y-coordinate position of the nodes in each stage is indicated by the order of the nodes within the stage, where the first node is the highest, and so on. Links are then drawn between appropriate nodes in adjacent stages, with the exception that if the node at which a link should terminate is not in the next stage, the dummy node for that node is, and the link is terminated here (without an arrow). If a dummy node appears in a stage, a link is drawn from it to the similar dummy node in the next stage (without an arrow), or to the actual node for which it is a dummy (with an arrow), whichever appears in the next stage. In the final layout, dummy nodes appear only as points.

With the graph underlying figure 3-9a the following stages are formed during the first step:

$$\begin{aligned} S(0) &= (1) \\ S(1) &= (2, 3) \\ S(2) &= (4, \boxed{5}, 6, \boxed{7}) \\ S(3) &= (5, 7) \end{aligned}$$

The next step of the algorithm then consists of rearranging the order of nodes within the stages in such a way that intersections are minimized. First an $N \times N$ matrix, P is formed where

$$N = \sum_{i=1}^m |S(i)|. \text{ The rows and columns are labeled with the}$$

names of the nodes in each stage, in order, starting with those in $S(0)$, then $S(1)$, and so on. The matrix entry $P(a,b)$ is then zero unless there is a link from node a to node b , in which case the entry is one.

Once P is formed, the number of intersections which would result from drawing the layout according to the order within each stage may be seen in P . For, one intersection occurs in the layout for each case in which, for $x < y$, $i < j$ (where $a < b$ implies that row (and column) a precedes row (and column) b in P), both $P(x,j) = 1$ and $P(y,i) = 1$, and only in these cases. This is proven as follows:

Lemma: In the construction of the matrix, P , and the layout corresponding to this matrix, for each case in P for which both $P(x,j) = 1$ and $P(y,i) = 1$, where $x < y$ and $i < j$, there is exactly one intersection in the layout, and intersections occur only in these cases.

Proof: We first show that for each intersection, there is a corresponding case in P .

Let (x,j) and (y,i) be two links that cross. Since links essentially run only between two adjacent stages, it must be that $x,y \in S(n)$ and $i,j \in S(n+1)$ for some n . Furthermore, since the links cross, we must have that $x < y$ in $S(n)$ and $i < j$ in $S(n+1)$ or that $x > y$ in $S(n)$ and $i > j$ in $S(n+1)$. Assume the former (with no loss of generality). Then it must be that $P(x,j) = 1$ and $P(y,i) = 1$ for $x < y$ and $i < j$.

We now show the converse: if $x < y$, $i < j$, $P(x,j) = 1$ and $P(y,i) = 1$ then there is an intersection. Let it be the case that $x < y$, $i < j$, $P(x,j) = 1$, and $P(y,i) = 1$. Now, if $x \in S(n)$ and $y \in S(n+k)$ for some $k \geq 0$, then, since links only travel for one stage, $j \in S(n+1)$ and $i \in S(n+k+1)$. If $k \geq 1$ then by the row ordering in the matrix, $i > j$; this is contrary to the premise; hence, $k = 0$ and $x,y \in S(n)$ and $i,j \in S(n+1)$ for some n . Then x will be positioned above y in $S(n)$ and i above j in $S(n+1)$ yielding an intersection when the links are drawn.

QED

Thus the removal of all such arrangements in the matrix would

remove all intersections.* If it is possible to remove them, they may be removed by changing the order of nodes within a stage, and thus the order of the corresponding rows and columns in P . Once we have obtained a new P matrix (and corresponding node order in the stages) for which the constraint is not violated, the corresponding layout will be intersection-free.

An example might clarify this procedure. Suppose we have the stages:

* In the paper describing this algorithm DiGiulo and Tuan require that the matrix P meet two constraints if the layout is to be intersection free:

- i) The non-zero element of each row should be consecutively located, and
- ii) If the non-zero element of a row begins in column j , then no non-zero element of any previous row may begin in a column with column index less than j .

It is believed that this second constraint is in error, for this allows situations which violate the constraint stated in the text and proven to account for the presence or absence of intersections.

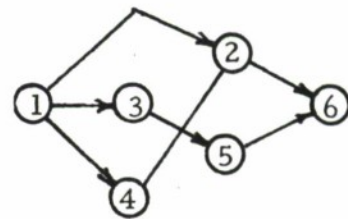
Furthermore, violation of the first constraint implies violation of the constraint in the text. This is shown as follows: let $P(x, k) = 1$, $P(x, k+k_1) = 0$, and $P(x, k+k_2) = 1$, for some x and k and some $k_2 > k_1 > 0$, and let $x \in S(n)$ (this is a violation of the first constraint). Now, since links travel only between adjacent stages, and by formulation of P , $k, k+k_2 \in S(n+1)$, and thus $k+k_1 \in S(n+1)$. But also by formulation of the stages, every element in a stage other than stage 0 has a predecessor in the previous stage. But obviously $n+1 \neq 0$, so that there must exist a $y \in S(n)$ for which $P(y, k+k_1) = 1$. Now if $y < x$ in $S(n)$, then since $P(x, k) = 1$, $P(y, k+k_1) = 1$ where $k < k+k_1$, the text constraint is violated. And if $x > y$ in $S(n)$, then since $P(x, k+k_2) = 1$, $P(y, k+k_1) = 1$ where $k+k_1 < k+k_2$, the text constraint is again violated.

For these reasons we have replaced the two constraints in the original algorithm of DiGiulo and Tuan, with the single constraint in the text.

$$\begin{aligned}
 S(0) &= (1) \\
 S(1) &= (\boxed{2}, 3, 4) \\
 S(2) &= (2, 5) \\
 S(3) &= (6)
 \end{aligned}$$

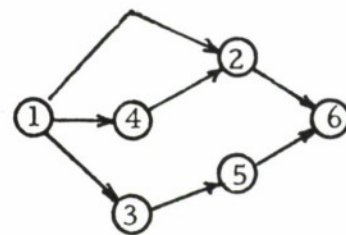
The initial matrix P is shown along with the layout it would generate in figure 3-10a. Note that the constraint is violated once in the matrix and one intersection appears in the graph. By exchanging rows and columns 3 and 4, we obtain the matrix of figure 3-10b,

P	1	$\boxed{2}$	3	4	2	5	6
1		1	1	1			
$\boxed{2}$					1		
3						1	
4					1		
2							1
5							1
6							



(a)

P	1	$\boxed{2}$	4	3	2	5	6
1		1	1	1			
$\boxed{2}$					1		
4					1		
3						1	
2							1
5							1
6							



(b)

Figure 3-10

which contains no constraint violations. The resulting intersection-free layout generated is shown adjacent to the new matrix.

It may not always be possible to remove all intersections, and certainly not in the case that the underlying graph is non-planar. However, the constraint optimization portion of network layout algorithms must attempt to meet this one constraint. Di Giulio and Tuan give no information on how they optimize, and no exact procedure will be given here. However, one observation will be made which allows us to outline such a procedure. We note that ones will appear only in certain submatrices of P , namely those covering both the rows of $S(n)$ and the columns of $S(n+1)$ for $n = 0, \dots, m-1$. All other entries in P are guaranteed to be zero. Let us number these submatrices $M(1), \dots, M(m)$ by column stage. Now, changing the column order in $M(n)$ affects two submatrices, $M(n)$ and $M(n+1)$, since we must also change the order of rows in $M(n+1)$. And, changing the row order in $M(n)$ affects two submatrices, $M(n)$ and $M(n-1)$, since we must also change the column order in $M(n-1)$. The remainder of P is unchanged by these modifications.

With this information in mind, we may then outline a method for proceeding through the matrix in an orderly fashion to remove violations of the intersection constraint:

- 1) Set $i = 1$ and go to step 2.
- 2) Check for violation of the intersection constraint in

submatrix $M(i)$. If there are no violations in $M(i)$, go to step 4. If there are violations, attempt to remove one or more by first changing column order in $M(i)$. If only changing column order does not suffice then row order in $M(i)$ may also be changed. If no violations can be so removed, go to step 4. If one or more violations have been removed, then, if only column order was changed, go to step 2; if row order was changed at all, go to step 3.

3) Set $i = i - 1$ and go to step 2.

4) Set $i = i + 1$. If $i > m$, exit; else, go to step 2.

This is only a suggestion for an approach to optimization. We do not claim that any decrease in the number of intersections will result. At this point we are unable even to guarantee that the procedure will terminate. The problem requires further examination.

A few additional difficulties are found with the procedure suggested by Di Giulio and Tuan which will be noted here. First, consider the situation for the underlying graph of figure 3-11a. If we follow the procedure given, we will be unable to remove the intersection shown in the layout of 3-11b. The procedure, as stated, requires that only one occurrence of a dummy node per real node exist in a given stage. Thus, effectively, links with common termination points are joined in the earliest stage possible. This treatment, as shown in 3-11b may generate unnecessary intersections. However, if we change the procedure to allow a number of

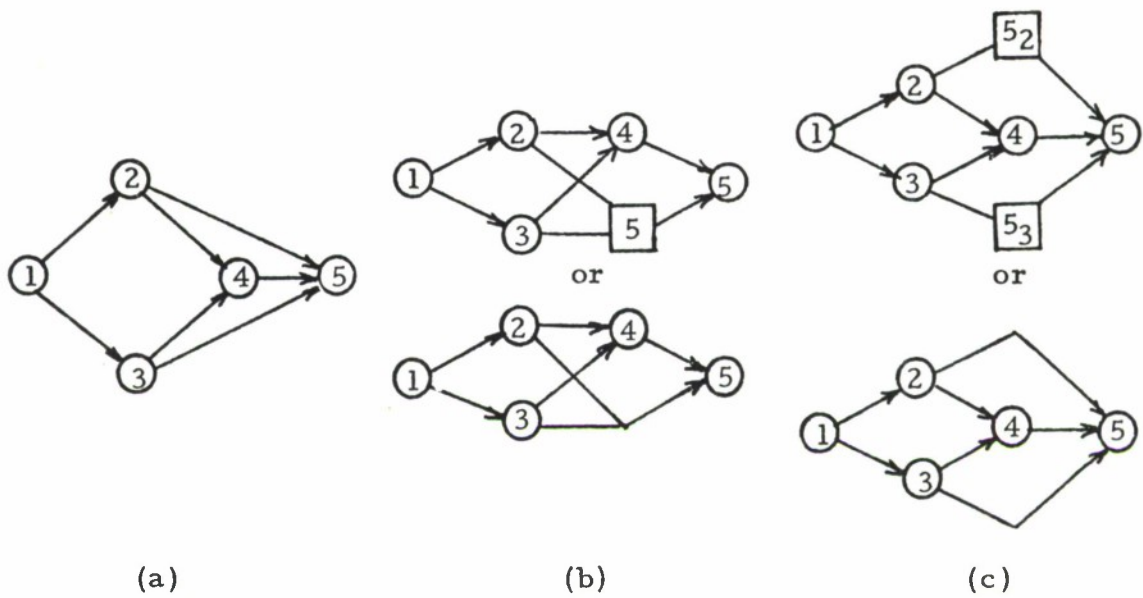


Figure 3-11

dummy nodes to be generated per stage, one for each predecessor, such unnecessary intersections may be avoided. For example, we might change step 2 of the stage generation process to read:

2) Form stage n , $S(n)$, from the union of all nodes y such that for $x \in S(n-1)$, there is a directed link from x to y . If the resultant $S(n)$ contains any node z found in a previous stage, say $S(n-i)$, $i \geq 1$, create q dummy nodes z_1, \dots, z_q in $S(n-i)$, one for each predecessor of z in $S(n-i-1)$, and remove z from $S(n-i)$. Delete all successors of z from stage $S(n-i+1), \dots, S(n)$, unless these nodes are also successors of other non-dummy nodes. Add the q dummy nodes, z_1, \dots, z_q , to each of the stages $S(n-i+1), \dots, S(n-1)$.

These q dummy nodes are to be handled in the P matrix in the same manner as any other nodes. However, when links are drawn, only dummy nodes with matching subscripts are attached. The q predecessors of the dummy nodes are attached only to dummy nodes with matching subscripts. When the real node represented by the q dummy nodes appears, all q dummy nodes are attached to it. Thus the stages for the example in figure 3-11a would be:

$$\begin{aligned} S(0) &= (1) \\ S(1) &= (2, 3) \\ S(2) &= (\boxed{5_2}, 4, \boxed{5_3}) \\ S(3) &= (5) \end{aligned}$$

The resultant layout is shown in figure 3-11c.

A similar difficulty is found with the procedure of Di Giulio and Tuan due to the requirement that non-dummy nodes be placed in the earliest stage possible. Consider the underlying graph of figure 3-12a, for example. According to the procedure of Di Giulio and Tuan, the layout of 3-12b would result, with one intersection. Again, the underlying graph is planar, and the intersection unnecessary. The procedural modification required here is more complex than that for the previous problem. What is necessary is a detection of situations such as that in stage 2, and a separation of such a stage into two stages, along with the creation of an appropriate number of dummy nodes. The results of such a modification on 3-12b are shown in figure 3-12c. The main problems here are the detection of the

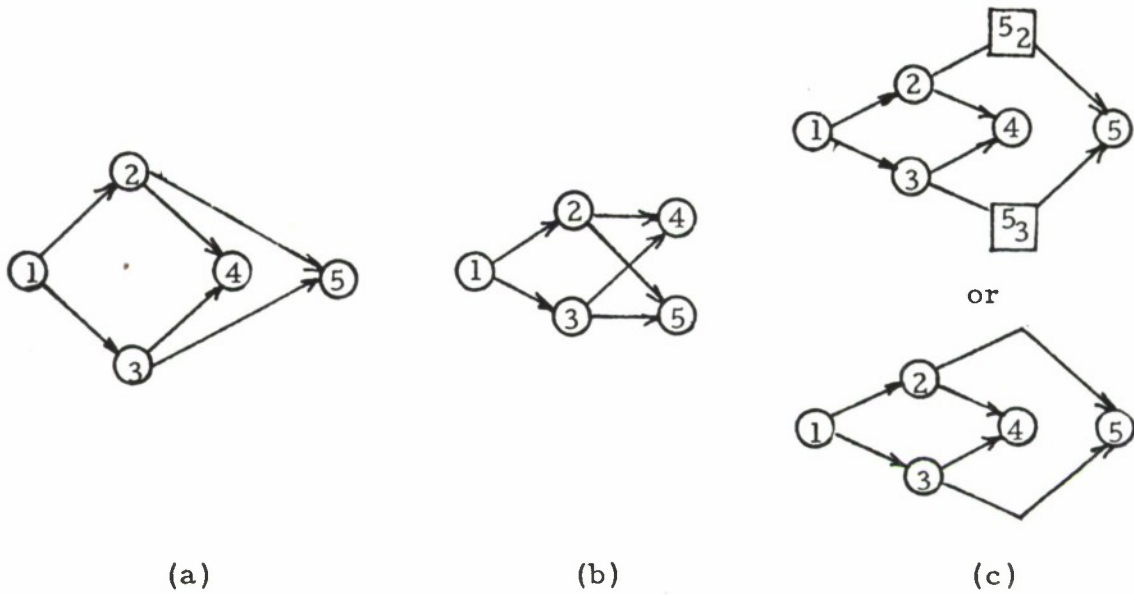


Figure 3-12

situation and the decision as to the manner in which the nodes of a stage should be broken up. For example, with the figure of 3-13a, no such solution is possible, whereas with 3-13b and c, a solution is possible, although it is difficult to make rules for obtaining these solutions.

Returning to our discussion of network layouts, we have seen that constructive algorithms required for their formation combine both structural rigidity and constraint optimization. In the last few paragraphs the interrelation between these two algorithmic characteristics has become apparent. For, it was shown that the more we allow the algorithm to optimize for minimal intersections, the less rigidly the resultant structure (and stage contents) could be predicted.

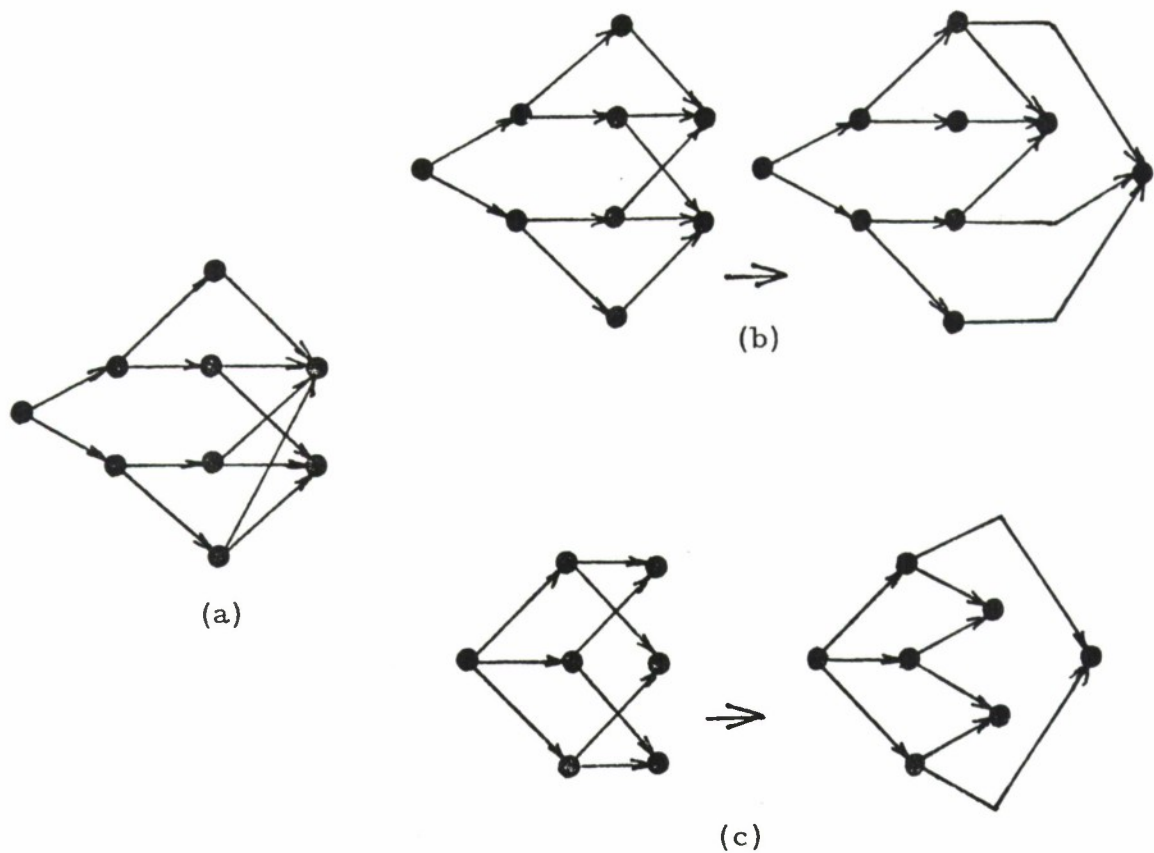


Figure 3-13

3.2.2.4 General Layouts

The main characteristic of the layouts classified under the general type is that very little, if any, particular structure is predictable. It is often the case that for these applications, the apparent underlying graphs used may be too complex for layout structure to be predetermined. We include here layouts with apparent underlying graphs which have all ranges of complexities and characteristics. The exceptions are those layouts which use ADP's or EP's.

The reason for this exception will become clear in the discussion of the next layout type. Algorithms for general layouts, then, must consist mainly of constraint optimization, as determined by the particular application. Subtypes may be differentiated by the particular constraints or qualities optimized, and the particular priorities the various qualities are given.

Among the layouts categorized under the general layout typing we find, for example, layouts used in graph theoretic illustrations, chemical structure diagrams, state diagrams, and layouts used as diagrams in countless other applications in which underlying graph structure is too complex for structural rigidity. We will consider a few of the layouts found in this typing, along with their constraints and priorities.

Layouts used to illustrate the chemical structure of a molecule, by convention, preserve certain familiar figures where possible, such as hexagons and pentagons. Nodes usually represent atoms and links represent bonds. In order that links be somewhat representative of the physical make-up of the molecule being depicted, link length consistency should be aimed at, along with a minimum number of bends. Another characteristic, not considered in section 1.1, and not common to many layout applications, the characteristic of three-dimensionality, is relevant in chemical layouts. What is meant here is that, where possible, the layout should

be drawn so that a three-dimensional interpretation may be given to the layout. Three-dimensionality is, in general, very difficult to realize, unless, of course, the three-dimensional coordinates of the nodes are known, in which case, the layout may be generated as two-dimensional projection of the three-dimensional object. This quality, however, is not the type of layout criterion considered in this work, and will not be treated here.

Layouts used as graph theoretic illustrations may be formed in many ways. Often the requirements of the layout are dependent on the particular idea to be illustrated. However, in general, the idea of greatest concern is that the graph theoretic structure be easily seen in the resultant layout. This suggests that the quality of fidelity, as described in section 2.1.1, might have first priority. Next in priority might be those qualities which add to directionality in layout such as minimum number of intersections, minimum number of bends, and minimum total link length. In fact, most of the qualities discussed in section 2.1 are appropriate for application here, and the decision about priorities may become quite arbitrary.

With state diagrams, we find that the qualities and priorities are similar to those of graph theoretic illustrations, although the most important qualities are probably a minimum number of intersections and a minimum number of bends. For, with state diagrams, the greatest concern is that links be easy to follow (and to label).

Again, most of the other qualities considered in section 2.1 are applicable to state diagrams. Examples of layouts from these three applications are shown in figure 3-14.

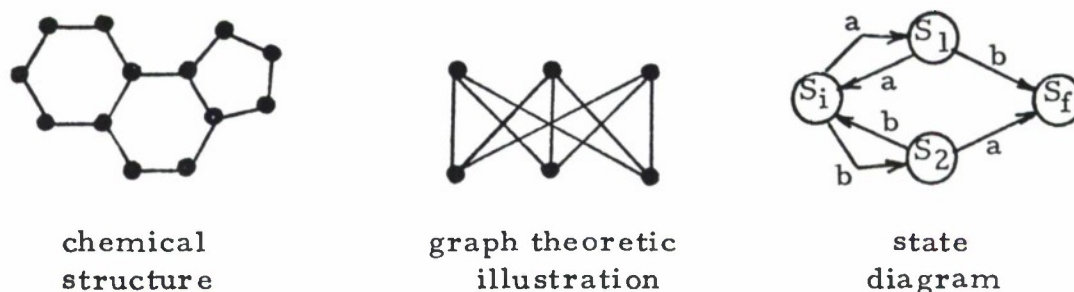


Figure 3-14

Thus we see that different applications are distinguished by different sets of qualities and priorities. A small amount of structural rigidity may be present. In the case of chemical structures, the requirement that, above all, certain familiar figures be realized, may be interpreted as a somewhat structurally rigid requirement. A chemical layout algorithm might begin by forming such figures, and throughout the constraint optimization phase of the algorithm, these familiar figures may be treated as inseparable units. However, with cases such as graph theoretic illustrations and state diagrams, the algorithms will be totally dependent on constraint optimization.

With this in mind, we must consider whether a sufficient number of the advantages of application dependency still remain with most layouts of the general type to merit the development of new

constructive algorithms for layout, in place of using sequences of modifying algorithms as in section 2.2.2. Although some structural rigidity may be present with general layouts, it is not enough, in general, to avoid the need for some arbitrary placement in layout. Furthermore, certainly, in many applications using the general layout type, algorithms which modify existing layout, rather than arbitrarily creating new layout, would be preferred. For these reasons, it is felt that perhaps, a better approach to layout for the general type is the modifying approach and that the constructive approach is improper for these layouts. We need not, however, lose the advantage of applying a single algorithm for layout rather than a sequence of modifying algorithms. For, since we know the quality priorities, we may combine the sequence into one algorithm. The modifying algorithms will be combined so that, in order that the changes they make will be overridden the least, those algorithms for qualities with higher priority will be performed later than those for qualities with lower priority.

An example might make the above discussion clearer. Suppose we are given the graph underlying figure 3-15a, along with its present layout. This figure represents a state diagram for a machine which accepts the strings $a^n b$ and $b^n a$ for n odd. We wish to minimize the number of bends and the number of intersections, above all else, perhaps, giving priority to the number of intersections. Instead

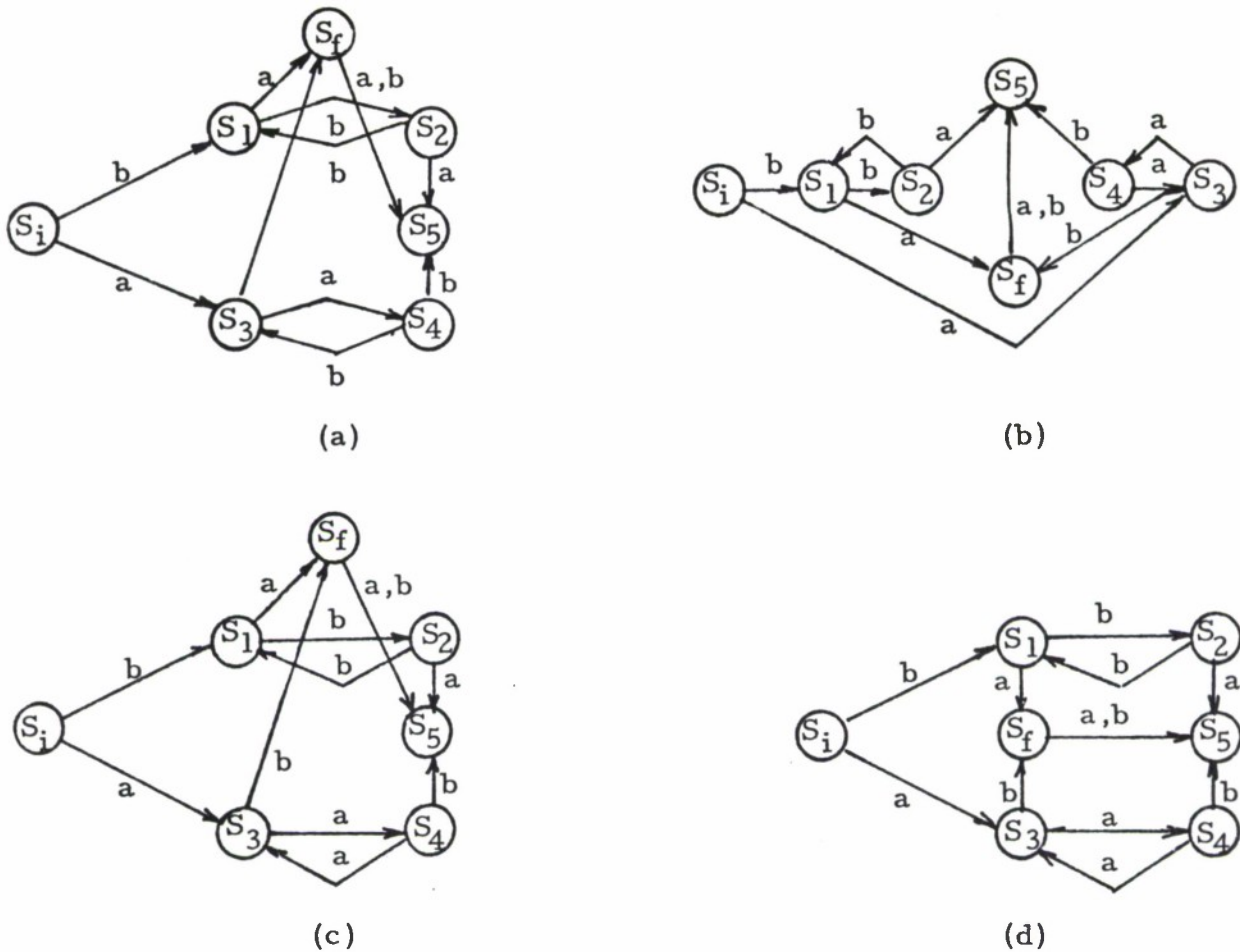


Figure 3-15

of using a constructive algorithm which ignores the initial layout, and which might give us the layout in figure 3-15b, we use an algorithm which is a combination of modifying algorithms of section 2.2.2, first one for bend removal, and then for intersection removal. This algorithm would first give us the layout of 3-15c, and finally that of 3-15d. This final figure depicts the structure of the state diagram in a layout which is more in accordance with the original layout than that of 3-15b.

In the cases where some structural rigidity is to be

maintained, as with chemical structures, two approaches are possible within this modifying framework. On one hand, a layout algorithm may consist first of a series of modifying algorithms applied in the order suggested above, followed, second, by a procedure which insures that the requirements of structural rigidity are met. Or, on the other hand, a layout algorithm may first introduce the structural rigidity, and follow this by an appropriately ordered sequence of modifying algorithms similar to those in section 2.2.2, but which have been modified so that in any manipulation performed, they retain the particular structural rigidity introduced in the first part of the layout process. It seems that since we are building algorithms for particular applications, perhaps, the second approach would give the best result. For, in a sense, all of the optimization is tailored to the particular application with this approach.

One problem remains, however, with the use of modifying algorithms in combination in a non-interactive environment. Suppose the original layout were drawn as in figure 3-16a. Then no intersection would need to be removed, although the number of bends is large. A single layout algorithm made from a combination of modifying algorithms might then yield the result in 3-16b, rather than that in 3-15d, since the bend removal algorithm must not cause any intersections to be created. But if we were to allow the user to interact with the layout procedure, rather than applying a

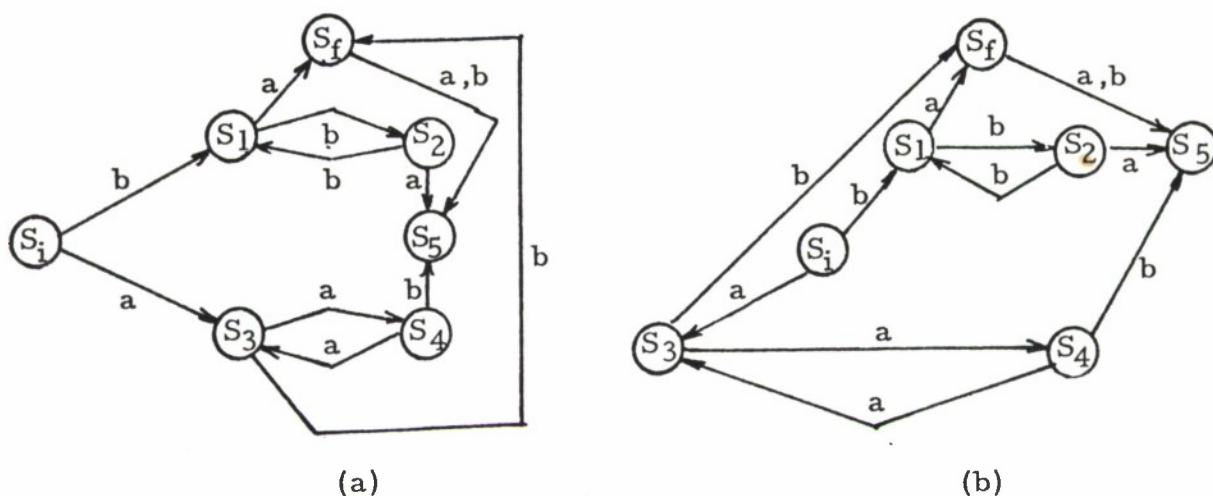


Figure 3-16

non-interruptible process, we could obtain the result of 3-15d, which seems to be a better layout in terms of clarity, and which is closer to the layout originally drawn. Thus a single non-interruptible algorithm for layout might not produce the best result, and, with algorithms containing so little structural rigidity, perhaps, some amount of user interaction is desirable.

Thus we might even question whether a single algorithm approach is suitable for most applications using general layouts. Or, in fact whether, due to the non-predictable structure of their underlying graphs, the interactive type of environment used in chapter 2 is best. Even when some structural rigidity exists, as in the familiar figure requirement of chemical layouts, placing specially designed modification algorithms which retain initially introduced rigidity in an interactive environment, might prove to yield

better results than those provided by a single non-interactive process.

3.2.2.5 Ordered-Arc Layouts

The ordered-arc type includes layouts which are found in applications where there may be little structural rigidity, due to the complexity of the apparent underlying graphs, as with the general type. But these layouts also contain specific fixed points on their nodes at which links may terminate or begin (EP's or ADP's). Hence, there is some restriction of the order in which links may be arranged around a node. The reason for the differentiation of layouts of this type from those found in the general type is that, due to this extra restriction on links, additional difficulty is found in developing optimization procedures for some qualities. Any movement during an optimization process, which changes the order of links around a node must be examined to avoid violation of these order restrictions.

The best example in which the effectiveness of an optimization procedure is inhibited by this restriction is found with the modifying algorithm of section 2.2.2.7 for minimizing the number of intersections in a layout. The algorithm proceeds by moving nodes from one region to another. Often, when a node is moved in this way, the link order around some node is changed. In fact, Anger (2)

states that the number of intersections for each representation of a graph with a distinct link order around the nodes is completely determined by this order. Thus, if link order is restricted, the minimal number of intersections for the underlying graph may not be attainable in the layout.

For example, consider the layout of figure 3-17a. If we restrict link order to that which appears, we cannot obtain an intersection free layout; whereas, if we allow order to change around nodes b and d, we may obtain the intersection free layout in 3-17b.

We see then that the intersection elimination algorithm which has been designed for layouts in which link orders may be changed, may not be effective with ordered-arc layouts. Other algorithms for these layouts must be designed. Similar problems are seen with many of the algorithms discussed in section 2.2.2. For example, those algorithms given for the realization of repetition, familiar figures, link length consistency, fidelity, bend removal, and

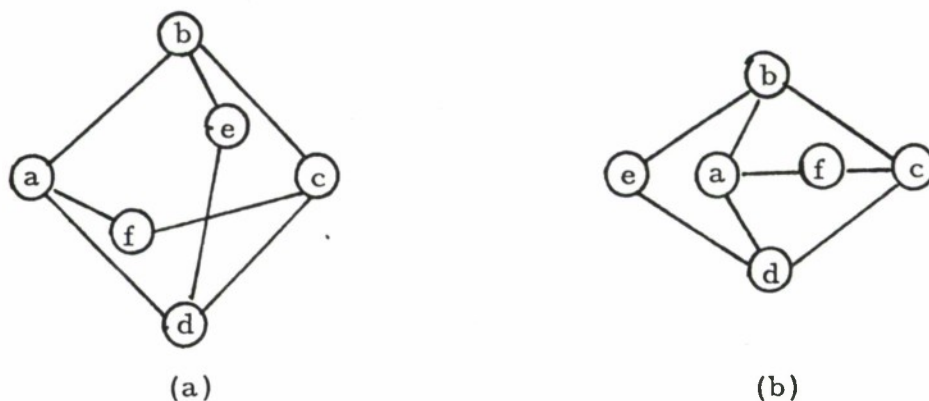


Figure 3-17

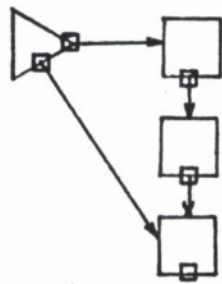
minimum link length may also change the order of links around a node.

Thus, although ordered-arc layouts are similar to general layouts, algorithms cannot, in general be built on the modifying algorithms of section 2.2.2. Instead we will see that specially designed optimization procedures must be used for these layouts. These include both modifying and constructive or semi-constructive algorithms. The fact that constructive algorithms have been built for the layout of some ordered-arc layouts, whereas we find this most difficult with general layouts needs some explanation. With ordered-arc layouts, the restricted link ordering may impose structure on a layout which has an effect similar to structural rigidity, although, here, the basis for the structure is inherent in the original layout given rather than imposed on the layout by the algorithm, as with structural rigidity found above. For, the link order itself may limit the number of ways the layout may be formed. Thus there is not as much room for arbitrary positioning as there is with layouts of the general type.

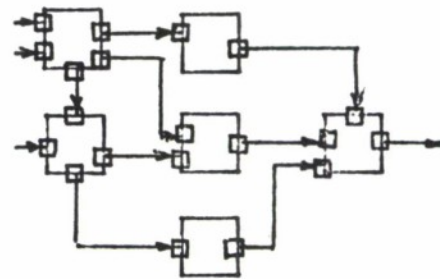
The problem of designing either modifying or constructive algorithms for ordered-arc layouts is very difficult. What has been done in many cases is that the layout problem has been broken into subparts. For example, first the node positions might be decided, and then the links might be routed. Only a few algorithms developed

to date treat the nodes and links of ordered-arc layouts simultaneously.

Among the applications in which ordered-arc layouts are found we will consider two in particular, AMBIT/G and circuit layout. With AMBIT/G layouts, only the points of link origin are fixed; links may terminate anywhere on a node. Whereas, with circuit layouts, where the nodes represent circuit elements with fixed input and output pins, both link origin and termination points are fixed. Example of these layouts may be seen in figure 3-18.



AMBIT/G Graph



Circuit Layout

Figure 3-18

Little work has been done on automatic layout generation for AMBIT/G. The most important quality for layouts of graphs output in this particular application is that the layout of an output graph resemble, as closely as possible, the layout of the input graph from which it was derived. Here, again, we have a layout criterion which is common to few applications and has not been considered in section 2.1. After this first criterion has been met, most of the criterion

found in section 2.1 are applicable. As mentioned above, however, the algorithms for realizing these qualities must be changed in order to account for the presence of ADP's. An additional restriction also occurs in AMBIT/G, in that nodes may not change their orientation; for example, they may not be turned upside down. Criteria may take new forms as a result of such restrictions. For example, examine the criterion used in the original AMBIT/G output program. This criterion causes a node at which a link terminates to be placed in the direction of the link relative to the node at which the link begins.* Such a criterion tends to minimize the number of bends, and, in general, to add to directionality. Yet such a criterion is not possible with general layouts, since neither are the points at which links leave a node fixed, nor, in general, are the node orientations.

Thus an algorithm for AMBIT/G layout must first base the layout on that of the original input graph, and then fill in the remainder using a procedure which optimizes the qualities of section 2.1, but which is specifically designed to deal with the additional restrictions of ADP's and node orientation. The amount of arbitrary placement resulting will depend on the similarity of the output graph to the input graph, and upon the amount of restriction in link order provided by the ADP's in the particular output graph. In a sense,

*See chapter 1.

we may consider such an algorithm as semi-modifying due to the fact that it is based on another layout.

With circuit layouts, both link origins and termination points are fixed, although node orientations, in general, are not. Circuit layouts are used to represent the manner in which circuit elements are to be placed and wired in manufactured circuits. Two design and economic considerations determine the most important criteria for circuit layouts. These are that the number of intersections be minimal and that the total wire (link) length be minimal. The reason for the first criterion is that circuits are often printed; for each intersection, a special bridge must be built to insulate one wire from another, and each such bridge adds to the manufacturing cost for the circuit. The second criterion is relevant for two reasons. First, the more wiring necessary, the higher the cost for the circuit, and second, the greater the length of a wire, the longer it takes for a signal to go through it. Such delay is undesirable in circuits. Few other criteria of sections 2.1 are relevant to circuit layouts, since such layouts are not, in general, intended for human consumption.

The problem of automatic circuit layout, given these two criteria, has been studied for many years, and has been of increasing interest in the field of computer development. The problem has been called the "backboard wiring problem" in the past. Many

methods for layout have been suggested, and will be summarized here. We will see that the development of an effective constructive algorithm which treats the complete problem is very difficult, although several attempts have been made. Instead the problem is usually broken up into a node placement stage, "the placement problem," and a wire routing stage, "the connection problem," and each is treated separately. Each of these two problems assumes the other to be solved, or, to be solvable, and ignores it. It has been suggested by Breuer (9) that this divided treatment of the problem is not as effective as a total solution might be.

Several authors have considered variations of the problem which are relevant to circuit layouts, but which go beyond the scope of our study. Among these variations are the concept of orthogonal wiring, wiring on two sides of a board to which elements are attached, one side with horizontal wires, and one with vertical. Also considered is multi-layer wiring, or placement and wiring on several layers or boards. Since our goal is to study the layout problem on a two-dimensional surface, we will not concern ourselves with these variations.

Let us begin by examining some of the approaches developed for the placement problem. Most approaches solve the problem on a grid. The general placement problem may be stated as follows: Given a set of elements $E = \{e_1, e_2, \dots, e_k\}$ and an $m \times n$ grid, P ,

where $m \times n \geq k$, with positions P_{ij} , $i = 1, \dots, m$, $j = 1, \dots, n$, find a placement of E on P such that a function F dependent on the placement, is minimized. A more restricted form of the problem has been considered by Miehle (27), in which some of the elements are in fixed positions.

The function, F , to be minimized is usually a measure of the optimization of a particular placement for wiring purposes. Thus, for example, F may be a measure of the total resulting wire length or the number of intersections in the wiring of the system or both. The connection information for the system, which is needed for the calculation of F , is usually recorded in a $k \times k$ connection matrix, C , where the entry C_{ij} is the number of wires connecting e_i and e_j .

A common approach to the problem is the modifying approach in the form of a one or two step algorithm, in which, starting with an initial placement, each pass through the algorithm improves the function F by a change in the placement. The algorithm is considered completed when no improvement can be made by another pass; in other words, a local minimum has been reached. The minimum reached is, in general dependent on the initial placement, as with most modifying algorithms.

As mentioned in section 2.2.2.8, Steinberg (33), whose only aim is to minimize wire length, formulates a solution as a one-step algorithm. His algorithm has been used as the basis for many

others. In preparation for the application of the algorithm, a family of unconnected sets $\{U_1, \dots, U_p\}$ is formed from E , such that every $e_i \in E$ is contained in at least one unconnected set, and if e_i and e_j are in the same unconnected set, then $C_{ij} = C_{ji} = 0$. An initial placement is then specified. A step of the algorithm proceeds by considering one of the unconnected sets, U_s , and the set of positions A_s which are either unoccupied, or occupied by $e_i \in U_s$. It is then necessary to find that placement of U_s into A_s , leaving all other elements fixed, which minimizes the function F . Steinberg defines F as the sum of a function, g_i :

$$F = \sum_{i=1}^k g_i; \quad g_i = \sum_{j=1}^k f(C_{ij}, d_{p(i), p(j)})$$

where $d_{p(i), p(j)}$ is a function of distance depending on the placements of e_i and e_j , $p(i)$ and $p(j)$, respectively, and where $f(C_{ij}, d_{p(i), p(j)}) = 0$ if $C_{ij} = 0$. We have then that:

$$F = \underbrace{\sum_{\substack{i \notin U_s \\ j \notin U_s}} f}_{F_1} + \underbrace{\sum_{\substack{i \in U_s \\ j \notin U_s}} f + \sum_{\substack{i \notin U_s \\ j \in U_s}} f}_{F_2} + \underbrace{\sum_{\substack{i \in U_s \\ j \in U_s}} f}_{F_3}$$

Now F_1 remains fixed for all placement of U_s in A_s since none of these elements are moved. F_3 is zero since $C_{ij} = 0$ for all $e_i, e_j \in U_s$. And finally, it remains to find that placement of U_s

into A_s which minimizes F_2 . But this problem may be formulated as the classical assignment problem as defined by Flood (15): given N men and M jobs, $N < M$, and for each man-job combination, a rating a_{ij} , $i \in N$, $j \in M$, find the assignment of men to jobs that minimizes $\sum a_{ij}$. In our problem, U_s replaces N , A_s replaces M , and a_{ij} is the function $g_i = \sum_{k \notin U_s} f(c_{ik}, d_{j, p(k)})$ where $p(i) = j$. Steinberg refers to the work of Munkres (30) and Kuhn (23) for solutions to the assignment problem. The algorithms discussed in these papers consider the matrix $A = \|a_{ij}\|$, and look for a set of N independent entries (one in each row and no two in the same column of A), such that the sum of this set is a minimum. Given the solution to the assignment problem, Steinberg's algorithm proceeds by placing the elements of U_s in the optimal positions determined, and concludes the step by calculating the value of F resulting from this new placement. The algorithm proceeds cyclically through U until, for p successive steps, no improvement is made in F .

In his work, Rutman (32) makes several improvements and additions to Steinberg's basic algorithm, which make it more suitable for computer use and which better the results. He gives a procedure for forming a family of unconnected sets, and a method for solving the assignment problem efficiently on a computer using Munkres' algorithm.

Rutman also mentions that often an interchange of connected

elements helps to improve the solution. In Steinberg's algorithm, no means for such interchange is provided, as connected elements are never processed in the same step. Thus Rutman suggests that interchanges be made at certain intervals between algorithmic steps. Finally, he states that in Steinberg's algorithm groups of tightly connected elements are reluctant to move; as a result, long single wire connections tend not to be minimized in some cases. To remedy this, he suggests that the algorithm proceed in two phases. In the first phase, by changing the definition of the function g_i , longer wires tend to be minimized. In the second phase, all wires are minimized by another function, g_i . The results Rutman obtains using these modifications tend to be better than those from the unmodified algorithm.

Steinberg's algorithm as formulated has not considered the problems specific to ordered-arc layouts, even though he is dealing with this layout type. However, this is to be expected, since neither does he concern himself with the number of intersections the resultant layout will contain. He may thus ignore link order since changing the link order, while keeping link terminals fixed, essentially affects only the number of intersections in a layout.

In another general approach to the placement problem, a constructive approach, the elements are placed one at a time on the board. The position and time at which an element, say a , is placed,

is based upon the positions of elements already placed, and the relation of element *a* to these elements.

Gamblin, Jacobs and Tunis (16) use this method to place those elements with "maximum conjunction" and "minimum disjunction" of pins in closest proximity. They choose from those elements not yet placed, one which ranks highest in this respect to the elements already placed, and a position is then given to this element. With this positioning the authors conclude that smaller wire lengths and fewer intersections will result when the system is wired.

Case, et al. (11) also incorporate this element-by-element approach into their placement system. For each element in turn, where order here is arbitrary, that position is chosen which minimizes a distance measure with respect to the elements already placed. The measure used is the sum of the rectangular distances from this element to all others to which it is connected and which are already on the board. However, at the conclusion of this initial phase, the placement measure is improved by a series of element interchanges between every pair on the board. If the placement measure is improved by a given interchange, the new positions are kept. The process continues until no gain is obtained by such interchanges. This process does not concern itself directly with minimal intersection, however.

As an interesting sidelight, we mention the solution formulated

by Miehle, as a system of equations, the solutions of which are found by known numerical techniques. Miehle has used this approach to find a placement for a subset of E , where the remaining elements of E have fixed positions, such that wire length is minimized. The set of equations which he formulates is a mathematical description of a physical model he built. With pins as elements, where some are fixed and others freely movable, the connections are modeled by the winding of a string throughout the system. When the string is pulled, the movable pins assume those positions which minimize total string length.

In summary, there have been two main approaches to placement, the modifying approach used by Steinberg, and the constructive approach used by Gamblin, et al. In the former, no consideration is given to the minimal number of intersections, whereas in the latter, both minimal intersections and minimal length are kept in mind.

We now consider approaches to the second part of the back-board wiring problem, the connection problem. The connection problem may be stated most generally as follows: given a list of connections to be made between fixed points on a two-dimensional surface, specify the routes of these connections so as to minimize a function F , dependent on this routing. As mentioned in section 2.2.2.7, algorithms which solve such a problem are considered to be semi-constructive, since part of the layout is fixed (the nodes), and part is

to be constructed (the links).

The problem then, for circuits is to find routes for wires which minimize the total length and the number of intersections. This problem has been examined by Lee (24). Lee's algorithm provides a general method for finding a path from one point to another on a grid, while minimizing any number of monotonic functions simultaneously. A function, f , is monotonic with respect to Lee's algorithm if for every path $p(c^i, c^j)$ we have the inequality $f(p(c^i, c^k)) \leq f(p(c^i, c^j))$ where $p(c^i, c^k)$ is any subpath of $p(c^i, c^j)$. For the back-board wiring problem, the functions considered are, of course, total length and number of intersections, although Lee gives examples of other functions, such as minimal proximity to certain objects.

Generally speaking, the algorithm proceeds step-by-step on a grid, building outward from the starting cell of a path, until the final cell is reached. If the final cell is not reached after a certain point, no path exists. With each step those cells adjacent to already recorded cells, which are not barriers to a path and which maintain a minimum value for the series of functions being considered, are recorded, so that when the final cell is reached, the minimal path may be traced back to the starting point.

More specifically, let steps from a given cell be allowed in four directions: up (\uparrow), down (\downarrow), left (\leftarrow), and right (\rightarrow). Let the set L be the set of those cells from which we may still explore paths,

and to which paths are still minimal. If we are to find a path from A to B, originally L contains only A. Let us also assign to each cell in L a cell mass, or set of minimal values for the series of functions under consideration; this appears in the form of a vector of values. The cell mass for A is a vector of zeroes. The algorithm is then:

- 1) Form L_1 , a set of all cells adjacent to those cells in L which are not barriers. For each of these cells find the minimum function value sums possible, and the direction of the cell in L from which a step was taken in order to obtain the minimal sum for this cell. Go to step 2.

- 2) Choose that set of cells from L_1 with the minimum function value sum; add this set to L , and assign those function values as cell mass for these cells; delete L_1 , and go to step 3.

- 3) Adjust L by removing those cells in L , all of whose adjacent cells, if not barriers, have had their cell mass determined.

When B appears in L , the minimum path may be obtained by tracing back from B according to the directions recorded. If L is exhausted before B appears, no path exists.

A small example should clarify the algorithm. Consider the board shown in figure 3-19 where cell 11 is a barrier cell.

1	2 A	3	4
5	6	7 X	8
9	10 X	11 F	12
13	14	15 B	16

Figure 3-19

Let us consider optimizing two functions, f and g where:

$$f(p(c^i, c^i)) = 0$$

and for all $c^j \neq F$:

$$f(p(c^i, c^j)) = \begin{cases} f(p(c^i, c^k)) + 1 & \text{iff } c^j \text{ is } \uparrow, \downarrow, \leftarrow, \text{ or } \rightarrow \text{ of } c^k \\ \text{undefined, otherwise} \end{cases}$$

$$g(p(c^i, c^i)) = 0$$

for all $c^j \neq F$ and $c^j = X$:

$$g(p(c^i, c^j)) = \begin{cases} g(p(c^i, c^k)) + 2 & \text{iff } c^j \text{ is } \uparrow, \downarrow, \leftarrow, \text{ or } \rightarrow \text{ of } c^k \\ \text{undefined, otherwise} \end{cases}$$

and for all $c^j \neq F$ and $c^j \neq X$:

$$g(p(c^i, c^j)) = \begin{matrix} g(p(c^i, c^k)) \text{ iff } c^j \text{ is } \uparrow, \downarrow, \leftarrow, \text{ or } \rightarrow \text{ of } c^k \\ \text{undefined, otherwise} \end{matrix}$$

Then, originally, the cell mass of A or cell 2 is (0,0) and the set $L = \{2\}$. Proceeding to step 1 we find L_1 to be:

$$L_1: \begin{matrix} & & \text{sum} \\ 1 & (1,0) \rightarrow & 1 \\ 3 & (1,0) \leftarrow & 1 \\ 6 & (1,0) \uparrow & 1 \end{matrix}$$

since, for example $f(p(2,1)) = 1$, $g(p(2,1)) = 0$. In step 2 we find the minimum function value sum to be 1, and thus we choose the set $\{1,3,6\}$ to add to L . We now assign those function values to 1,3, and 6 as cell masses which are shown in figure 3-20, and we can

1 $(1,0) \rightarrow$	2 A (0,0)	3 $(1,0) \leftarrow$	4
5	6 $(1,0) \uparrow$	7 X	8
9	10 X	11 F	12
13	14	15 B	16

Figure 3-20

delete 2 from L , leaving $L = \{1,3,6\}$ in step 3. Proceeding through

steps 1, 2, and 3 again, we have:

			sum
L_1 :	5	(2,0) ↑	2
	10	(2,2) ↑	4
	7	(2,2) ↑	4
	4	(2,0) ←	2

The minimum sum is 2 and thus the set {5,4} is added to L, and each of these cells is assigned a cell mass. Cells 7 and 10 are left unassigned. L finally becomes $L = \{3,6,5,4\}$. Continuing for four more passes through the algorithm we have the steps shown in figure 3-21.

\underline{L}_1	<u>sum</u>	<u>min. sum</u>	<u>cells assigned</u>	<u>resultant L</u>
9 (3,0) ↑	3			
10 (2,2) ↑	4			
7 (2,2) ↑	4			
8 (3,0) ↑	3	3	9, 8	{3, 6, 9, 8}
7 (2,2) ↑	4			
10 (2,2) ↑	4			
13 (4,0) ↑	4			
12 (4,0) ↑	4	4	7, 10, 13, 12	{10, 13, 12}
14 (3,2) ↑	5			
16 (5,0) ↑	5	5	14, 16	{14, 16}
15 (4,2) ←	6	6	15	{15}

Figure 3-21

We have then reached B with the cell masses shown in figure 3-22.

We may trace the minimum path back to A, to obtain the resultant path 15-14-10-6-2. Note that several arbitrary decisions were made in assigning cell masses and directions, and thus, that this is not a

1 $(1, 0) \rightarrow$	2 A $(0, 0)$	3 $(1, 0) \leftarrow$	4 $(2, 0) \leftarrow$
5 $(2, 0) \uparrow$	6 $(1, 0) \uparrow$	7 X $(2, 2) \uparrow$	8 $(3, 0) \uparrow$
9 $(3, 0) \uparrow$	10 X $(2, 2) \uparrow$	11 F	12 $(4, 0) \uparrow$
13 $(4, 0) \uparrow$	14 $(3, 2) \uparrow$	15 B $(4, 2) \leftarrow$	16 $(5, 0) \uparrow$

Figure 3-22

unique minimal path.

Lee's algorithm has been incorporated into many backboard wiring systems. Case, et al, for example use their own heuristic for wiring a board, but follow its application by the application of Lee's algorithm to complete paths which their own heuristic is unable to complete.

In applying Lee's algorithm, however, the question of order of routing arises. Breuer (9) says that when longer wires are routed first, more channels are blocked, making it difficult to route short wires, whereas, since longer wires are more difficult to route, perhaps they should be placed first. Thus, one may either minimize the number of wires left unrouted or minimize the total wire length of unrouted wire.

Vincent-Carrefaur (37) has tried to do away with ordering in the process of routing connections by forming all paths simultaneously. However he finds this "dynamic" method too inefficient for large system. In exploring several algorithms for simultaneous routing he finds the "obstacle method" to give the best results. With this method, initially all connections are made along the best paths possible. Each path is then examined and adjusted with respect to the others, until no gain is made by the adjustment. If a required minimum is not met, then the least optimal path is removed and replaced with a path which was previously eliminated but is more optimal. Although good results were obtained with this method, large systems are not handled easily.

Thus we find that Lee's algorithm contains the basic approach underlying many solutions to the connection problem, by routing wires using a search for the best path with respect to a given set of functions on the paths. Both criteria for circuit layout may be optimized in the result.

Let us now examine the approaches in which the two problems are handled together. Breuer (8), who maintains that this approach is the best, since the two problems are closely interrelated, has formulated the two problems together as an integer linear program. He forms a set of inequalities which require that the elements be placed on a rectangular grid with no two elements occupying the

same position. In his formulation, Breuer allows that links are not completely determined, but that certain groups of pins which are to be connected are specified. He then asks that for each such group the tree with the shortest length, which connects members of the group be used, and that the sum of all of these shortest lengths be a minimum. He also allows other restrictions to be specified, for example, that certain elements be directly connected, that two elements be separated by a certain distance, or that no connection be longer than a certain length.

Kalish (20), in his MAPID system, which is unusual in that he considers ease of reading as one criterion, also handles the two problems together. His system consists of two processes, a "topological" process and a "geometric" process. The topological process is subdivided into two phases. In the first phase, the "horizontal" phase, the elements are placed in separate columns in order of dependence, with each element placed to the right of all its input elements. Single-input-single-output elements are not considered. The vertical positions for elements are assigned in a snake-like pattern with each element in a different row as well as column.

In the second phase, or "vertical" phase, the elements are adjusted vertically in their columns and connections are rerouted to minimize intersection. All connections are restricted to a simple

L-shape. Initially a list of intersections is made. The list is then processed cyclicly. A step consists of processing the intersection at the top of the list. In an attempt to remove the intersection, the four nodes which it involves are moved vertically in their columns above and below all connections in the columns. The position with the total net gain in number of intersections removed is chosen. If there is no gain, the intersection is placed at the bottom of the list. Any new intersections are recorded. When the list has been processed completely at least once with no net gain, the process ends and the geometric process begins.

The geometric process is intended to increase clarity. It begins by the inclusion of those single-input-single-output elements discarded in the previous process. These are placed in separate columns but in the same row as the horizontal part of the connections on which they lie. There are then two phases of the geometric process: a "vertical" phase and a "horizontal" phase. In the vertical phase the elements are moved up or down in their columns to obtain a minimum number of rows and a minimum number of bends in the connections. In the horizontal phase, to reduce unnecessary space, all elements are moved to the left as far as possible, and then those elements with more output than input are moved to the right to reduce the number of long wires. We notice that his approach is similar to that used for network type layouts.

Another approach to the simultaneous solution of these two problems is found in the work of Mamelak (26) who is concerned with the layout of logic diagrams. He tries to identify "chains" or sets of interconnected elements, two of which are connected to all of the remaining elements of the set. Mamelak claims that these chains are characteristic of circuits. Each chain found is assigned to a row or column of the board. In this way intersections are minimized. Placement of chains relative to one another is determined by total wire length, and other electrical criteria. He claims that the results obtained using this method were comparable to the work of experienced designers.

We notice that with Kalish's approach to the problem, a means for structuring the layout has been chosen which is not part of the convention of circuit layout. In fact, the structuring chosen reduces the problem to one similar to that for network type layouts. This cannot be considered a general solution for layouts of the ordered-arc type since such a structuring may not always be possible. In Mamelak's solution another structuring is chosen, but it is one which is based on what the author believes to be a characteristic of graphs underlying this type, and hence, seems a more natural solution. Breuer's approach, however, simply states the requirements and searches for a solution without any prestructuring. Since no structure is assumed, it seems that solutions found

using his system would be the most effective. The problem with his approach, and, in fact, with many of the approaches mentioned, is that large circuits are difficult to handle.

In summary, many solutions to layout for the ordered-arc type have been developed. No single algorithmic form stands out as being most effective, although many have been tried. All involve a great deal of optimization and little prestructuring, and must, in general, be tailored to the particular peculiarities of the application. For this reason we consider ordered-arc type the most complex.

3.2.2.6 Summary of Layout Types

We have classified several layout types common to many applications, and considered algorithms for their layout. Layout types were ordered, and a gradation was found in algorithm characteristics and complexity corresponding to this ordering. The simplest type, linear layouts, may be formed using a totally structurally rigid algorithm which is constructive in nature.

In the next type, tree layouts, a gradation in algorithm characteristics appears according to subtype, although constructive algorithms were found for all subtypes. With all leveled subtypes and with the unleveled subtypes with bends, again a totally rigid structure is predictable. However, with the unleveled, son-centered, subtype without bends, the predictable structure is not as

rigid, and with the unleveled, level-centered subtype without bends, again, the structure is not as rigid and optimization is required for the first time in the classification.

In the next type, network layouts, again constructive algorithms are feasible. A good amount of structural rigidity is still present, but optimization is also required to a great degree.

With general layouts we find that little may be predicted about layout and most of the layout depends upon optimization. Due to the complexity and unpredictability of their underlying graphs it was felt that constructive algorithms are not appropriate for layouts of the general type but that the realization methods discussed in section 2.2.2 should be used as the basis for these layout algorithms. It was found that, perhaps, the best environment for layout of the general type is an interactive one.

Finally, ordered-arc layouts were discussed. These layouts whose underlying graphs are essentially as complex as those for general layouts, have the additional difficulty of the restriction of link order. It was found that algorithms are as complex as for general layouts, but that they cannot be built on those methods developed in section 2.2.2 in most cases. In order to form layouts of the ordered-arc type, special optimization algorithms must be developed which account for the special restrictions of this type. Realization of qualities for these layouts is, in general, more difficult than with

layouts of the general type, and algorithms tend to be more specialized, difficult to perform on large layouts, and often promise less than optimal solutions.

Hence, we see that knowledge of particular application characteristics may or may not be helpful in layout. With the simple layout types, constructive algorithms are easy to develop and to apply, and the results may often be guaranteed to some extent. With increasing complexity in type, difficulty in developing algorithms approaches and even goes beyond that found with the general case discussed in chapter 2. But, given this framework for layout type classification, we may see where in this range a new layout type falls. Thus, this classification may be useful in developing the simplest layout algorithms possible for a new layout type, and in understanding what factors contribute to the complexities encountered in its layout.

3.3 A DESIGN FOR THE EXTENSION OF MOD

In this section we will briefly consider changes and extensions to the MOD system, which allow us to take advantage of the type dependent information discussed above. Here, we are concerned mainly with modification of the MOD Output system, since provision has already been made in the Input and Framemaker systems for graph layouts of many types. Our aim is then to somehow allow the

user to input, along with his graph layout, information concerning layout type, and to have the system provide him with a type dependent menu of algorithms (either modifying, constructive, or both) which he may use to obtain a new layout.

Such a design is easy to envision as an extension of the present system. For, as the system now is designed, it contains a small library of modifying algorithms which may be easily extended. Rather than having a single library, we might give the user a choice of several libraries, containing both modifying and constructive algorithms. The choice could depend on layout type as specified by the user. The contents of a library would be a set of algorithms tailored to a particular layout type. The user will have a choice of algorithms, as he does now.

Let us consider what a sample session might be like with an extended MOD Output system. Initially, the user might find the frame shown in figure 3-23 on the scope. He may then input, output, or manipulate the current graph layout as in the old MOD Output, or, he may choose one of the layout types listed in the column on the right.

When a choice is made, the menu in the right-hand column would change to a list of commands for the execution of the algorithms in the library for this particular type. This list would also include a label, "new type." Pointing to this label returns the user to the

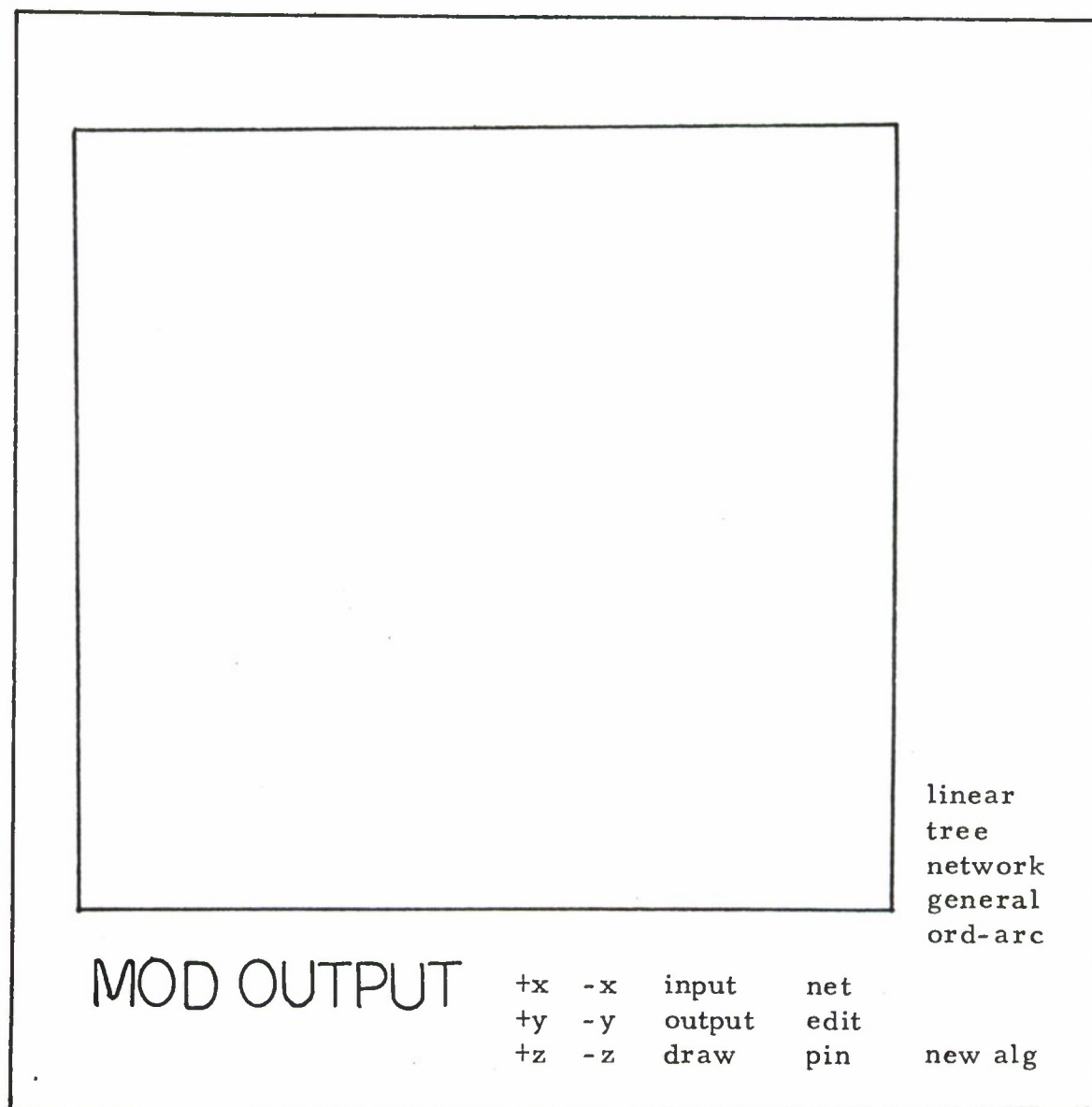


Figure 3-23

type selection found in figure 3-23, while retaining the graph layout in the box. For example, should the user choose the tree type he would obtain the frame of figure 3-24.

In addition, when a new type is chosen, the system checks that, if a graph layout appears in the box on the scope, it conforms to the restrictions of the particular type chosen. For example, when the tree type is chosen, the graph layout must not contain cycles. When network is chosen, the graph layout should be directed and contain no directed cycles.* When these conditions are violated by the existing graph layout, the words "graph layout does not conform to type" will appear in place of the graph layout in the box. To remedy the situation, the user may either read in a new graph layout to be checked, keeping the type fixed, or, press "new type," to return to the original frame and choose a new type, retaining the graph layout.

When a graph layout and a type which agree have been chosen, the user may proceed to apply to the graph layout, any of the algorithms listed in the right-hand column. He initiates an algorithm by pointing to the appropriate label in the menu. If the algorithm is a constructive one, the underlying graph structure is derived from the

* Or, perhaps some means can be built for the user to specify partial node order when no arrows appear. This problem also exists with the linear graph type, in that node order must somehow be indicated.

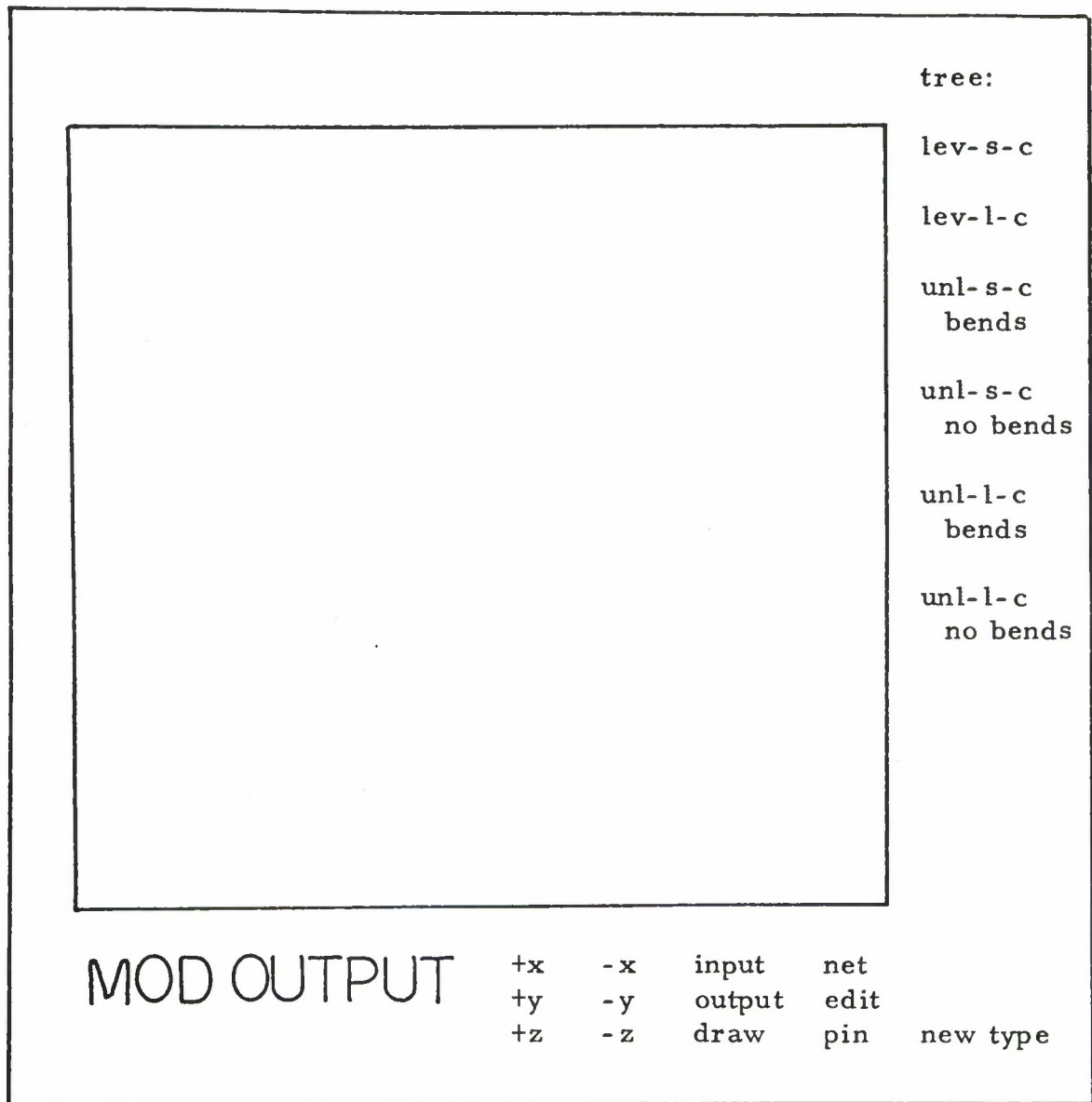


Figure 3-24

graph in the box, but the present layout is ignored and eventually lost. If the algorithm is a modifying algorithm, it would use the present layout as a basis. As many algorithm applications as desired could be performed.

When the user has modified the layout to his satisfaction, using the set of algorithms for a given type, he can then output the new layout as in the old MOD Output system, read in a new graph layout within the same type or within a new type, and so on. He may, if his graph layout is appropriately structured, process it under several types.

It is necessary with some of the algorithms, that further information be supplied for their execution besides that given by the graph layout in the box. For example, when processing a tree type layout with one of the algorithms for the tree type, the root node must be designated.* Thus, for example, with each execution of such an algorithm, a message might be typed as follows: "point to root node." In response, the user must indicate which node is to be treated as the root. Such information will be called "run information," and must be supplied each time the algorithm is executed.

Another type of information which we shall call "option

* The one exception might be when we input a tree which has a unique node with no incoming links. In this case we might assume that the root node is that unique node.

information" must be supplied to some of the algorithms. This information tells the algorithm more specifically what constraints the user desires, without having to list numerous versions of an algorithm in a type menu. For example, one of the tree type algorithms might ask for orientation, in the form of the typewriter message:

Type desired orientation:

- 1) root on top
- 2) root on bottom
- 3) root to left
- 4) root to right

Option information differs from run information in that, with the former, once a choice has been made, we may wish that choice to apply to several graph layouts without having to repeat the choice; but the latter is valid for only one application of the algorithm and must be specified each time the algorithm is executed. This difference will become important later in the discussion.

The libraries for the various types would initially be set up to correspond to the needs of the types as discussed in section 3.2. The initial plan for the contents of each of the libraries is as follows:

- 1) linear type: constructive algorithm for linear layout,
- 2) tree type: the various constructive algorithms for layout of tree layout subtypes,
- 3) network: constructive algorithm for network layout with provision for interactive adjustment of stages and creation of additional dummy nodes between steps,

4) general type: modifying algorithms of the type discussed in section 2.2.2; special modifying algorithms for specific cases such as chemical molecules, and

5) ordered-arc type: modifying and constructive algorithms, as well as semi-constructive algorithms, for ordered-arc layouts.

However, such a system is not complete without a good facility for creating and adding new algorithms. The "new alg" label in the menu of figure 3-23 is used for this purpose. There are two ways in which we would like to be able to create algorithms. First, it should be possible to add totally new algorithms to the system, and second, it should be possible to combine algorithms already included to form new algorithms. For example, we might wish to define a new algorithm for a tree layout subtype from an old one, with the only modification being that the orientation always be with the root on top. Or, we might want to write an algorithm for finding and forming hexagons and pentagons, to be added to an already existing algorithm for minimal number of intersections, thus resulting in a single modification algorithm for chemical molecules.

As mentioned in Appendix 5, it is not very difficult to program and add new layout algorithms to the old MOD system. It should be equally as easy to do this for the new MOD system, especially if the system is designed to treat the collection of algorithms simply as a list. To be included in the system, newly programmed algorithms

file			new alg:		
return					
erase					
edit					
copy					
linear:	tree:	network:	general:	ord-arc:	
.	lev-s-c	.	.	.	
.		.	.	.	
.	lev-l-c	.	.	.	
	.		A*		
	.		B*		
	.		.		
			.		
			.		

Figure 3-25

must be added before the system is run. Care must be taken, however, that the algorithm is appropriate for the type in which it is included, and, that its name be original. Algorithms added to the system in this manner, as well as those already in the system, are called "compiled" algorithms.

To combine algorithms in the system in order to form new

algorithms while the system is running, the user first points to the label "new alg" in the frame of figure 3-23. The frame of figure 3-25 then appears on the scope. This frame lists all of the algorithms currently in the system, under appropriate type. To form a new algorithm the user simply points to the label of an algorithm under its type and then to the place in the new algorithm list at which this algorithm is to be included. List order in the new algorithm reflects logical order. To delete an algorithm from the new algorithm list, he simply scrubs the name. Each addition to or deletion from the new algorithm is reflected in this list.

When an algorithm is included in the list, if it contains any option information inquiries, they are typed out as if the algorithm were really being executed. For each inquiry, the user responds on the typewriter with either a choice of options or with a carriage return. If an option is chosen, this choice will remain fixed in the new algorithm, and the option information inquiry will not be made when the new algorithm is executed. If the response was a carriage return, the option information inquiry will be included each time this instance of the algorithm is executed within the new algorithm. However, run information inquiries are always made.

Once the new algorithm designation is complete, the user points to "file." This action causes the new algorithm to be filed. A message is typed: "name," to which the user must respond with an original name for the algorithm, followed by a carriage return. If the

name is not original, the system again types "name." Once an original name has been given, the message: "type," is output. The user must respond with the layout type under which this algorithm is to be placed.

Upon completion of naming and typing, the algorithm will be filed under the type, and appropriately added to one of the type lists with a star next to its name. The star indicates that this is an "uncompiled" algorithm. All algorithms defined while the system is running are uncompiled algorithms. When the system is terminated, uncompiled algorithms will be lost. Of course, both uncompiled and compiled algorithms may be used in building new algorithms.

Only uncompiled algorithms may be edited or deleted within this facility while the system is running. Editing and deletion of compiled algorithms must be done before the system is run. Thus, compiled algorithms are permanent in a sense, whereas, uncompiled algorithms are temporary, and are intended for experimentation only.

To edit or delete an uncompiled algorithm the user points to "edit" and then to a particular algorithm in the type lists. The algorithm name disappears from this list, and the algorithm itself appears in the new algorithm list. The user then treats this algorithm as if he were just building it. Once appropriate changes are made he must refile it, in order that it be remembered. When an edited algorithm is filed, no new name is requested, and it is filed under the old name.

Type must be respecified.

If the user presses "erase," the new algorithm column is emptied, and the building process may be started anew. If an old algorithm was being edited and was not filed, this action amounts to deletion. All algorithms built which included this algorithm are also automatically deleted.

An uncompiled algorithm may also be copied as a basis for another new algorithm by pressing "copy," and then by pointing to the name of the algorithm. This action is equivalent to repeating the actions used to build the copied algorithm, and is included only as a convenience. In order to return to the frame of 3-23, the user presses "return." Any unfiled new algorithm is lost. Any changes in type libraries resulting from new algorithm definition will be seen in the menus for each type.

The user is responsible for the effect of any algorithms he builds. He may combine algorithms from several types, and no check is made that the result is consistent with respect to type. A few guidelines are given, however. Constructive algorithms obliterate any previous layout whereas modifying algorithms do not. Thus if both constructive and modifying algorithms are included in a single new algorithm, the constructive one should precede the modifying ones. *

* Otherwise, any results from the modifying algorithms are lost once the constructive algorithm is executed.

Furthermore, algorithms for qualities of highest priority should be placed as late as possible in the new algorithm.

The details of the system will not be worked out here. However, we note that compiled algorithms will take some common form in which run information and option information inquiries will be designed in the form of lists attached to the body of the algorithm. When new uncompiled algorithms are formed, they will be defined by lists of other algorithms. Each compiled algorithm included in such a list will be accompanied by an appropriate vector, designating which option inquiries for the algorithm are to be made and which have been fixed. Upon execution of a compiled algorithm, all option inquiries are made as well as run inquiries. When an uncompiled algorithm is executed, each algorithm in the defining list is examined in turn. If the included algorithm is compiled, the vector associated with it is examined; all run inquiries are made but only those option inquiries not specified as fixed are made. If the included algorithm is an uncompiled algorithm, it is performed in the same manner as for the uncompiled algorithm in which it is included.

Figure 3-26 shows an example of this structure. Suppose we have two compiled algorithms, A and B. Suppose also that there is one run inquiry, x, and three option inquiries for A, each with the possible choices, a, b, or c, and no run inquiries and one option inquiry for B with the possible choices, a, b, or c. The uncompiled

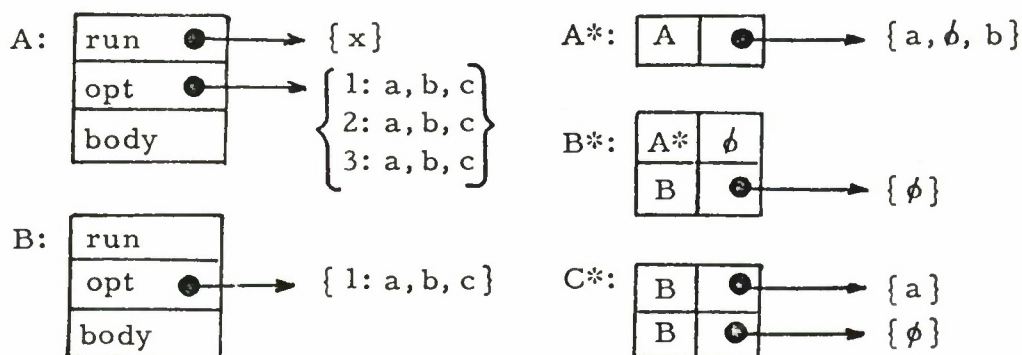


Figure 3-26

algorithm A^* then specifies, by a vector associated with its first and only entry, that A is to be run once with options 1 and 3 fixed to a and b , respectively, and that along with the one run inquiry, one option inquiry, namely 2, is to be made. B^* is designated by a list of two algorithms, A^* and B . With A^* no vector is needed and it is performed as above. B is then performed with a single option inquiry being made. Finally C^* consists of two executions of B , the first with no option inquiry and the second with one.

In summary, these proposed modifications to the MOD Output system allow the user to process a graph layout according to layout type, and provide a facility for the definition of new layout algorithms, both experimentally and in a permanent manner. The possibility of one further facility should be mentioned, that of allowing for new layout types. Such a facility should be included as a logical extension to this system. As with the addition of compiled algorithms, if the system is built to process the collection of layout types as a list, in some

orderly manner, this extension should be quite easy to make. All that is necessary is the careful addition of layout type requirements, a new type name, and any appropriately designed algorithms.

Chapter 4

ANOTHER APPROACH TO LAYOUT

When research was initially undertaken on the layout problem for graphs by this author, another approach to the problem was taken. This approach was found to be unsuccessful for several reasons. However, we will briefly discuss it here since it is a logical modification of the general layout problem and might be of interest at some future time.

This approach which we shall call the "modified layout problem,"^{*} has several variations. The basic idea is that, given a graph, its layout, and a list of changes to the graph (additions or deletions of nodes or links), a new layout for the modified graph should be produced. Several variations in the main constraint of the problem are possible. On the one hand, we may apply the constraint that as little be changed as possible in the original layout in obtaining the new layout (similar to the constraint for AMBIT/G graphs), or, on the other hand, we could apply the constraint that as little work be done as necessary in laying out the new graph. This second constraint is based on the assumption that it is easier to layout graph modifications within an existing layout than it is to layout the complete graph

^{*} This terminology is not to be confused with "modifying algorithms" as described in section 2.2.2. The two ideas are not directly related.

from scratch.

The motivation for using this approach was that the layout problem could be attacked gradually, as a graph was being built, rather than facing the problem with a completely specified graph. The approach using the first constraint certainly has value for AMBIT/G, and reduces to the problem defined for AMBIT/G layout which was discussed in section 3.2.2.5. With the second constraint, the question remains as to whether or not the amount of layout effort saved by algorithms for handling graph modification only, is worth the effort of developing them. In either case, we must also ask whether, in fact, such an approach provides any additional handle on the layout problem, or whether it really makes the possibility of solution more difficult.

In considering the approach using the second constraint, we have examined a sampling of what might be involved in providing a set of algorithms for handling graph modifications and resulting layout modifications while maintaining certain layout constraints. The modification of adding a single node to a tree layout of one of the subtypes discussed in section 3.2.2.2 was considered. As was shown in section 3.2.2.2, we may form these tree layouts according to certain constraints, and almost always obtain a result which conforms. The problem with adding single nodes to tree layouts is that, in general, unless some modifications of the original layout is made, the new tree

will not meet the constraints for the particular layout subtype. The only two exceptions to this rule are when a leaf is added to an existing leaf in the leveled, son-centered subtype, or when a new root is added in any subtype. In these two cases we need not change other node positions to maintain the constraints.

Let us look in more detail at modifications and procedures necessary for maintaining constraints in some of the various tree subtypes when a single node is added:

1) For the leveled, son-centered subtype:

a) To add a node, a , to a leaf, b : As mentioned above, this does not affect the positions of the other nodes; but we must adjust the appropriate level, S_{i+1} , to account for the new node (where b is in S_i), in order that further additions may be correctly handled. Do this by replacing the $K_{j_k} = \#$ in S_{i+1} by a , where $K_j = b$ in S_i . The x -coordinate of a is the same as that of b . If there is no S_{i+1} , derive one appropriately.

b) To add a node, a to a non-leaf, b : in this case, all x -coordinates are affected; nodes in levels above that of b are affected since space must be made for the new node, a ; and all nodes in levels below that of b are affected, since their sons have been moved. To obtain the new placement, first, add the node a to S_{i+1} appropriately, where b is in S_i . In each succeeding S_k add a $\#$ appropriately for a . After these insertions have been made,

recompute the x-coordinates for all S_k .

c) To add a node, a as a new root: this operation should not affect the other nodes, but again we must adjust the S_k . Renumber all S_k as S_{k+1} , and set $S_0 = (a)$. The x-coordinate value for a is the same as that for the old root.

2) For the unlevelled, son-centered subtype with bends:

a) To add a node, a , to a leaf, b : this may cause readjustment in the levels of b and of some of b 's ancestors, although new x-coordinates need not be calculated. The x-coordinate for a will be that of b . The algorithm in figure 4-1 is required to adjust levels. In this case it becomes especially clear that reapplication of the original layout algorithm is simpler than providing and running an additional procedure for such a special case of modification.

b) To add a node, a , to a non-leaf, b : this addition should not change the level of any node, but the x-coordinates will have to be recalculated. The new node should be added to the last level and marked by a # in each level down to, but not including, that of b .

c) To add a node, a , as a new root: similar to the case for the subtype above.

3) For the leveled, level-centered subtype:

a) To add a node, a , to a leaf, b : the only x-coordinates it changes are those in the level above that of b . If b is in S_i ,

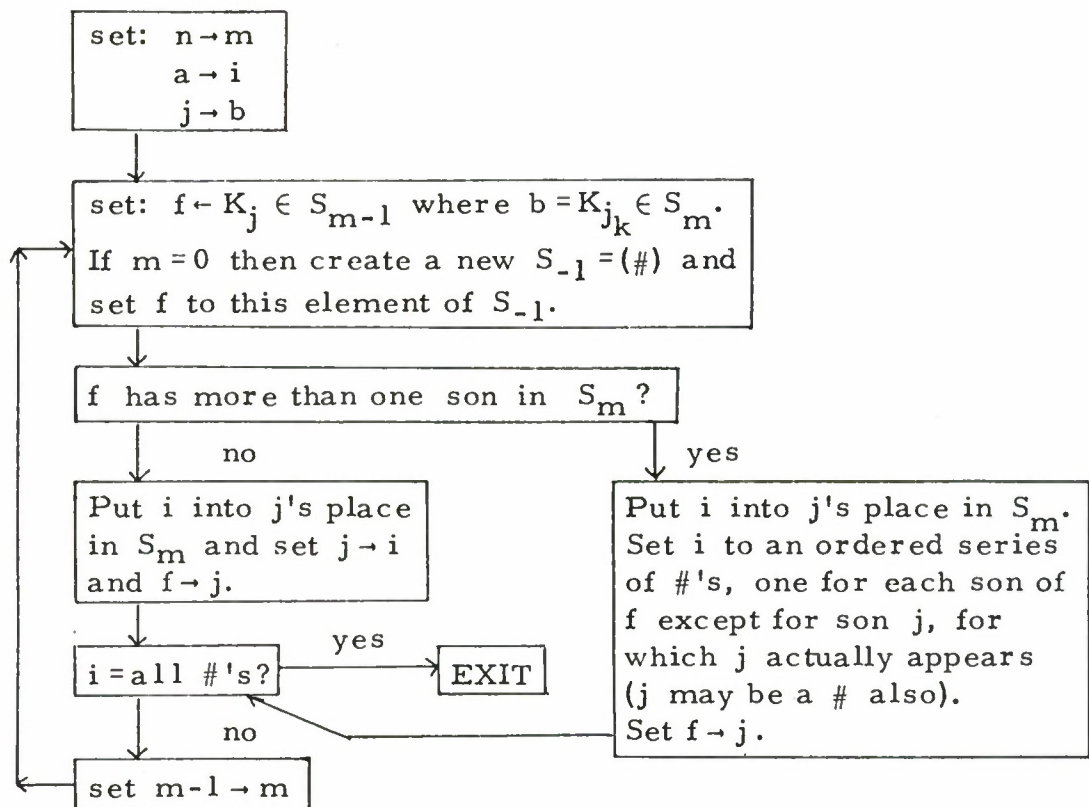


Figure 4-1

replace the # in S_{i+1} by node a as in the case for the leveled, son-centered subtype. Then recompute the x -coordinates in level S_{i+1} .

b) To add a node, a , to a non-leaf, b : this is similar in effect to the case for the leveled, son-centered subtype. Just add node a appropriately to S_{i+1} where b is in S_i , add #'s to the succeeding S_k 's, and recalculate the x -coordinates in S_{i+1} .

c) To add a node, a , as a new root: similar to the case for the subtype above.

4) For the unlevelled, level-centered subtype with bends:

a) To add a node, a, to a leaf, b: again, this causes the problems found with the case for the unlevelled, son-centered subtype. But here, x-coordinates must also be recalculated since the contents of the levels may change.

b) To add a node, a, to a non-leaf, b: add this new node and required #'s in the same manner as in the case for the unlevelled, son-centered subtype. The x-coordinates need to be recalculated for levels above that of b.

c) To add a node, a, as a new root: similar to the case for the subtype above.

In summary, all subtypes require some modification in the level contents in order to add new nodes. Six of these cases require a recalculation of x-coordinates within one or more levels, and two require redistribution of old nodes into levels, a very time-consuming procedure. But the most striking fact is that the numerous ways in which a node may be added must be differentiated within each subtype. Thus with tree layouts, certainly the reapplication of one of the original tree subtype algorithms would be much simpler than adding several new algorithms, and having to determine which is to be applied in each specific case.

It was concluded that this brief study was an indication as to what might be involved in more complex cases using this approach,

and work along this line was discontinued on the assumption that the effort was not worthwhile.

Furthermore, it was felt that such an approach does not deal with graph layouts in a manner suitable for effective constraint optimization. By nature it deals with graph layouts only in a local sense, at the points where graph modifications might cause change. The approaches used in chapters 2 and 3, however, are designed to handle graph layouts both in a local and a global manner, whichever is appropriate in a particular case. This leads us to the conclusion that this modified problem approach may result in algorithms which have limited power, and that the two approaches used in chapters 2 and 3 are much more effective in solving the problems of layout.

Appendix 1

AMBIT/G OUTPUT PROGRAM

Before describing the output program, we will first give a brief description of AMBIT/G itself. The language, as previously mentioned, deals with graphs. The data and program statements are both in this form. The graphs consist of specially designed nodes of various types and directed, labeled links. In the layouts of these graphs the nodes have shapes which are user defined and which include arc departure points (ADP's), the only points at which links may depart from the nodes. The nodes may also have names. The links are multisegment links. The program statements are in the form of graphs expressing patterns to be matched with the data and data graph changes to be made if a match is successful. The statements are labeled, and each statement contains two succeeding statement labels; one is used if the match is successful, and one is used if it fails. The node shapes, data graph, and program statements are drawn by the user. The program is then run on the data graph.

The output program is designed so that any portion of the current state of the data graph may be examined at any point during program execution. This is accomplished by the insertion of output statements in the program using the form shown in figure A1-1. These statements cause the specified portion of the data graph to be

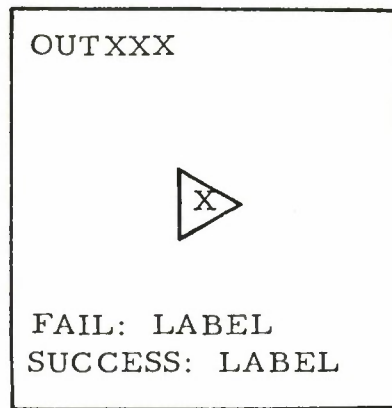


Figure A1-1

displayed; execution is stopped and the user has control of the scope until he presses the return button. At this point the display disappears and execution continues. The output statement succeeds if a display is generated, and fails otherwise.

As mentioned above, a super-structure must be built over the data graph for each display. It is described below. Several node shapes are reserved specifically for display purposes. These are shown in figure A1-2 where lozenges indicate the ADP's.

The output statement mentions a single node. This node must have a single link pointing to some display specification. The display specification is built of a series of pictures represented by P-nodes, the first of which is that pointed to by the node appearing in the output statement as in figure A1-3.

A P-node may point to a D-node giving this picture a directional placement in the total display, relative to the previous P-node or

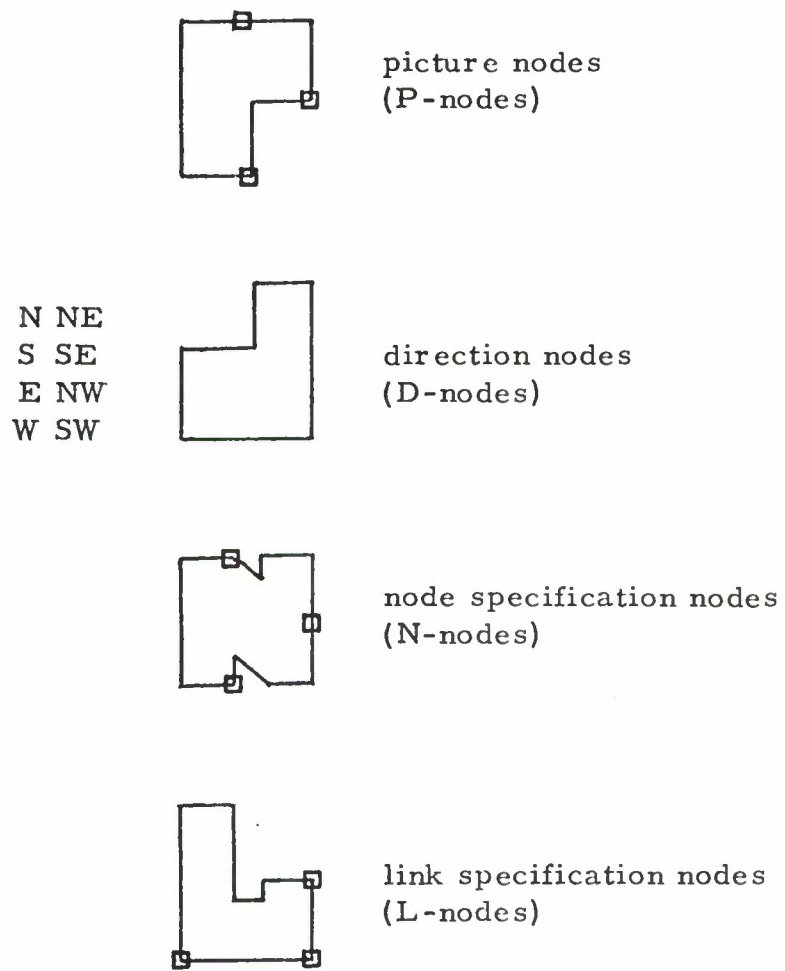


Figure 1-2



Figure A1-3

picture. For example, figure A1-4, will place the second picture to the right of the first in the display. If no D-node is specified, the current picture will be placed to the south of the previous picture.

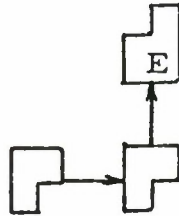


Figure A1-4

The contents of the picture for a P-node are specified by a string of L- and N-nodes, the first of which is pointed to by the P-node, as in figure A1-5.

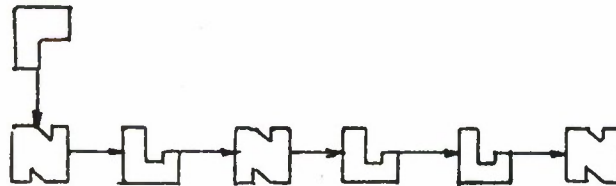


Figure A1-5

Each N-node in the string points to some data node which is to be displayed. Two N-nodes may point to the same data node, in which case this node will appear twice in the display. The N-nodes are examined in order, and, using D-nodes, direction may be assigned to an N-node, giving it a directional placement relative to the previous N-node. For example, the specification in figure A1-6 will result in

the picture of figure A1-7. When no direction is assigned to a node relative to the previous node by D-nodes, directional placement is

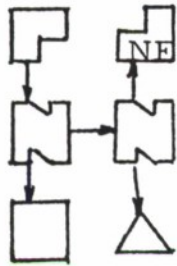


Figure A1-6

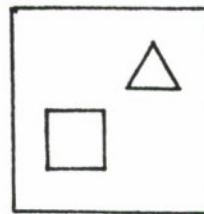
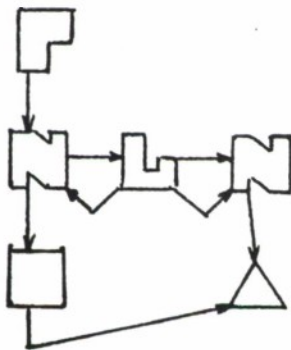
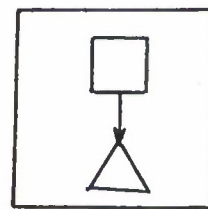


Figure A1-7

derived from link directions. For example, the specification of figure A1-8a results in the layout of A1-8b.



(a)



(b)

Figure A1-8

If no directional information is obtainable from the links a node is placed to the east of the previous node. For example, figure A1-9a yields A1-8b.

Each of the two bottom links from any L-node in the string must point to some N-node in the string. All links which originate

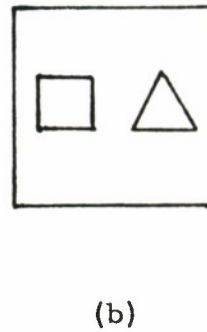
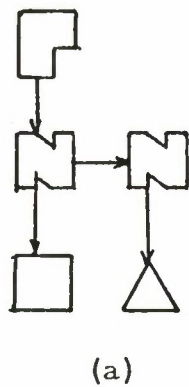


Figure A1-9

at the first data node referred to, and which terminate at the second data node referred to, are drawn. For example, figure A1-10a gives A1-10b. This super-structure must be correct in order for any display to appear.

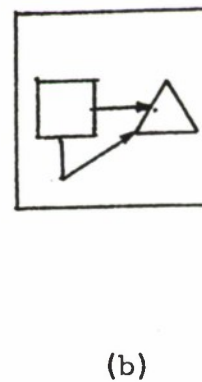
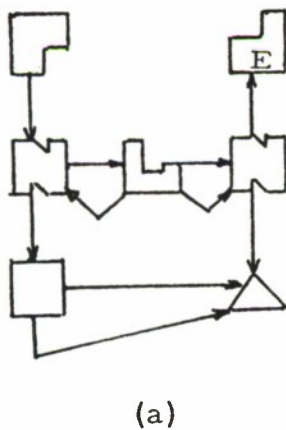


Figure A1-10

In addition to a simple display, the output program provides a facility for modifying displays which appear on the scope. As mentioned above, once a picture appears, the user has control of the scope until he presses the return button. It is while he is in control that he

may modify a picture. In brief, the modification facility allows the user to cause part or all of the graph layout to be moved or deleted. He may also enlarge or diminish the size of the graph layout. Furthermore, he may duplicate a node (but not its links, although links may be divided between instances of the same node), as well as add bends to links.

The actual layout algorithm used is described below. The nodes are placed on a grid, and links are drawn after all nodes have been placed. The procedure for node placement is as follows. First each P-node or picture is separated into subpictures using the steps shown in figure A1-11.

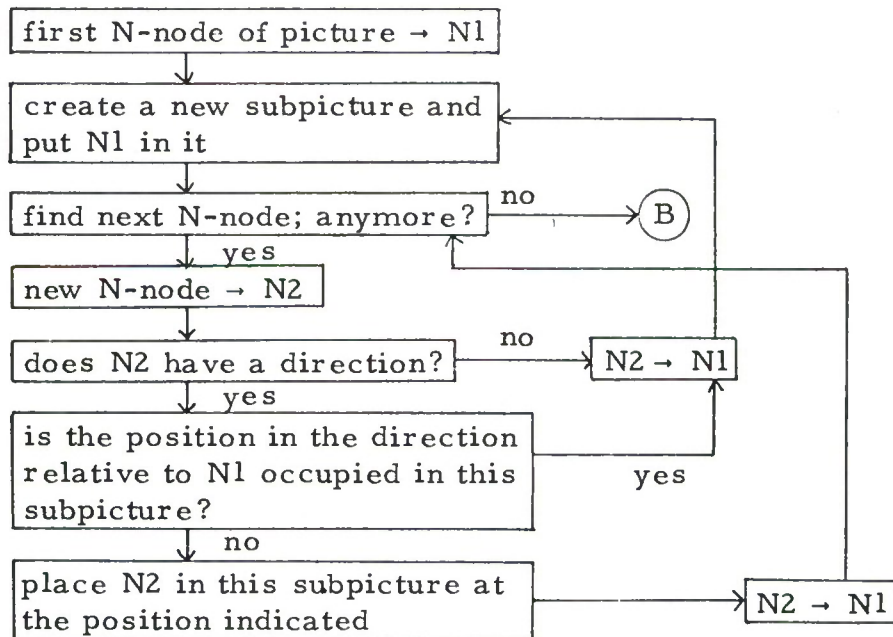


Figure A1-11

Then the positions for the subpictures are found relative to one another, as shown in A1-12.

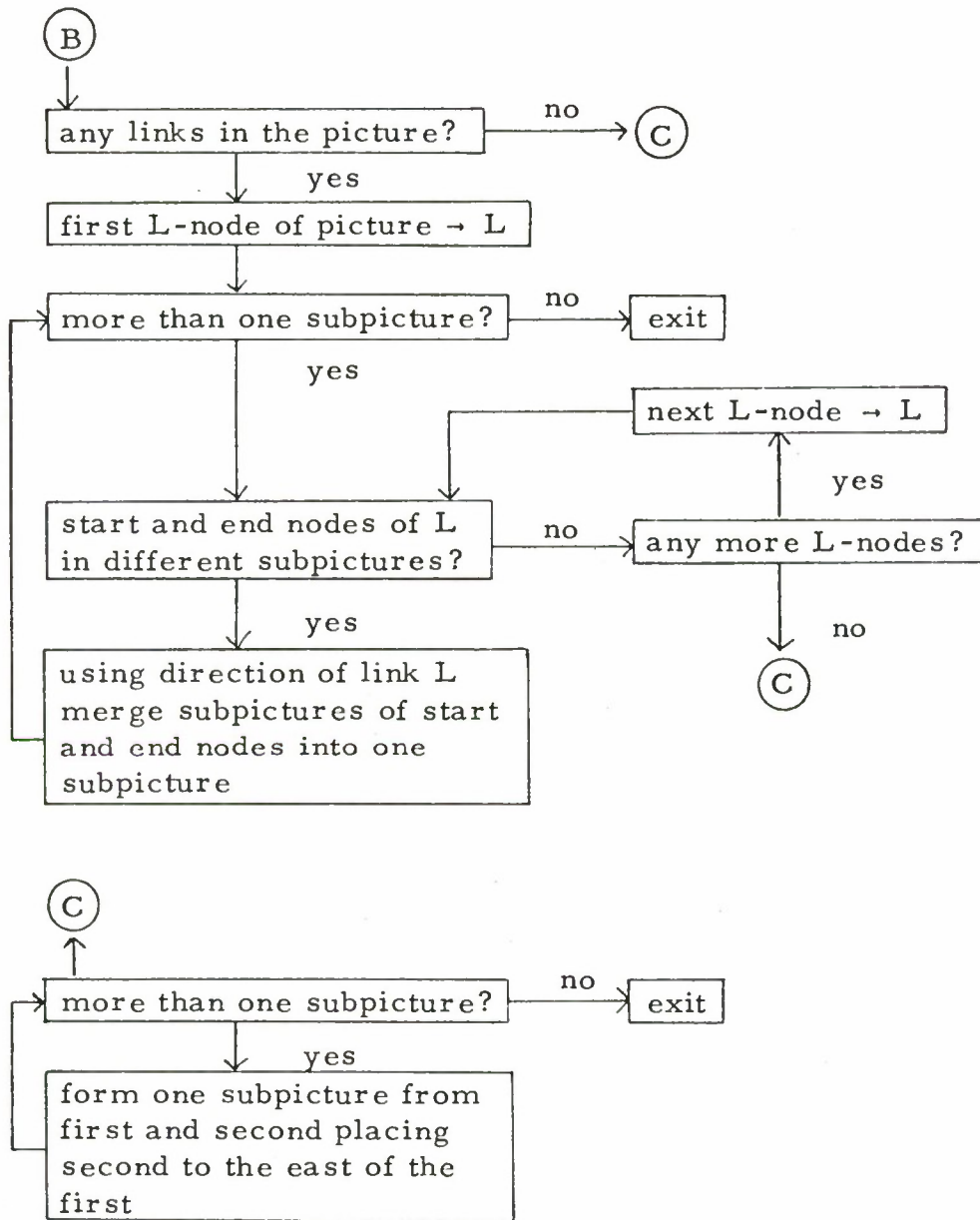


Figure A1-12

To obtain one display layout from several P-nodes or pictures we then follow the steps shown in figure A1-13.

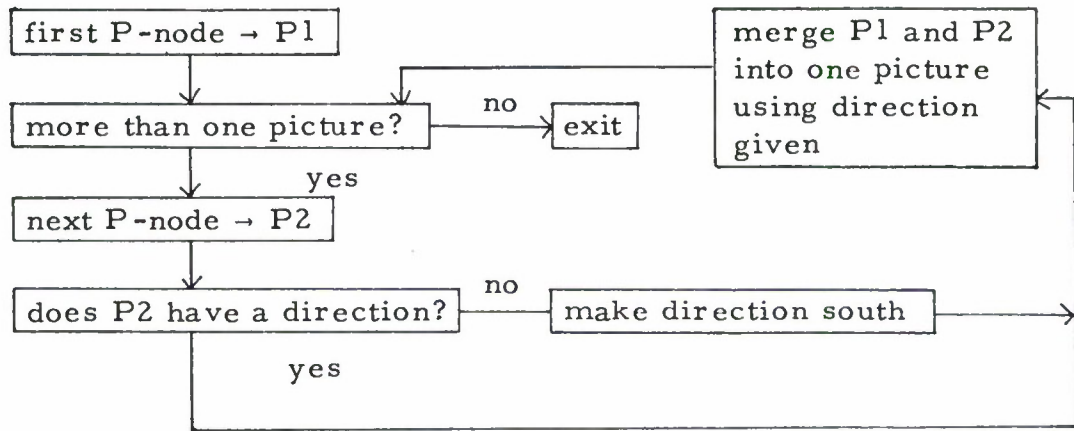


Figure A1-13

Once node positions have been obtained, links are routed between nodes. Links are routed as follows. A straight-line is begun from the ADP to the midpoint of the target link. If the line is obstructed by another node, the link is stopped just before the node, a bend is made, and the node is skirted until it no longer stands in the way. A straight line is again followed to the target node. All other obstructions found are skirted in the same manner until the link reaches its target node. The link end is clipped at the edge of the target node shape, and an arrowhead is drawn.

Appendix 2

AN ALGORITHM FOR REPETITION

Reading of this algorithm should follow reading of the discussion of section 2.2.2.1 of the text. The algorithm applies to a layout, two subparts of which have been designated by the user with appropriate enclosures. The effect of the algorithm is to change the layout so that the two subparts designated are made to be literal or symmetrical repetitions of one another, according to what the user desires.

In the description of the algorithm which follows the first subpart designated will be called A, and the second, B. When a subpart is designated, it contains all nodes which are enclosed, and all links connecting these nodes. If there is a bend in a link which is included, it will be treated as a node.* A center, $c(A)$, for a subpart, A, will be determined as follows, where, if p is a point, p_x implies its x-coordinate and p_y its y-coordinate, and where $a \in A$ implies that a is a node (or bend point) in subpart A:

$$c(A)_x = (\text{Max}(a_x, a \in A) + \text{Min}(a_x, a \in A))/2$$

$$c(A)_y = (\text{Max}(a_y, a \in A) + \text{Min}(a_y, a \in A))/2$$

The set S designates nodes common to both A and B, in other words $S = A \cap B$.

* Thus any isomorphisms between subparts must also include bend points. From this point on in the discussion, the word "node" is meant to include bend points as well as nodes.

The algorithm then proceeds through two main steps. In the first step, we look for isomorphic node-to-node maps between the two subparts (step 1a), and, in the case where symmetrical repetition is requested, we establish the axis of symmetry for each map (step 1b). The mappings are then ordered so that those which change the rest of the layout the least are used first (step 1c). In the second step, we attempt to make the two subparts literal or symmetrical repetitions of one another by either changing the second subpart specified to conform to the first (i. e., A cannot be moved), or by finding an intermediate form between the two subparts and changing both to this new form (i. e., A can be moved). We must consider the cases in which $S = \emptyset$ and those in which $S \neq \emptyset$ separately. All mappings are tried until a success is found, or the list is exhausted. Step 2 is divided into eight cases; only the appropriate case should be executed for each application of the algorithm.

The substeps of step 1 are then as follows:

(1a) Find all one-one mappings, f , of the nodes of A onto those of B such that for all $a, b \in A$:

$$b \in \Gamma(a) \Leftrightarrow f(b) \in \Gamma(f(a))$$

where $\Gamma(a)$ is the set of nodes adjacent to a .*

* Note that these mappings are isomorphic.

If the two subparts are to be made literal repetitions, we then remove from the set of mappings, all mappings:

- (i) f such that $f(S') = S'$ where $S' \subseteq S$ and $S \neq \emptyset$; this includes the case in which $f(x) = x$, $x \in S$, and
- (ii) f such that if there is a cycle in A labeled clockwise (a, b, c, \dots, x) , the corresponding cycle in B is not labeled clockwise $(f(a), f(b), f(c), \dots, f(x))$.

If the two subparts are to be made symmetrical repetitions, we then remove from the set of mappings, all mappings:

- (i) f such that if $x \in S$, $f(x) \notin S$,
- (ii) f such that $x, y \in S$, $f(x) = y$, $f(y) \neq x$, and
- (iii) f such that if there is a cycle in A labeled clockwise (a, b, c, \dots, x) , the corresponding cycle in B is not labeled counterclockwise $(f(a), f(b), f(c), \dots, f(x))$.

The justification for the removal of these mappings is given later in the discussion of the algorithm. The resultant set of mappings will be called F . If $F = \emptyset$ the algorithm fails.

(1b) If the two subparts are to be made symmetrical repetitions, determine the axis of symmetry α_f for each mapping $f \in F$, according to the size of S , and according to whether A can be moved or not, as follows:

(i) if $S = \emptyset$, α_f is the perpendicular bisector of the line $c(A), c(B)$.

(ii) if $|S| = 1$, and A cannot be moved, α_f is the perpendicular to the line $c(A), c(B)$ which passes through the single node in S .

(iii) if $|S| > 1$ and A cannot be moved, then α_f is the line which perpendicularly bisects all lines x, y such that $x, y \in S$ and $f(x) = y$, and which passes through all points $x \in S$ such that $f(x) = x$. If such a line cannot be drawn, no symmetrical repetition can result from this mapping; thus this mapping fails and is removed from F .

(iv) if $|S| \geq 1$ and A can be moved, α_f is the perpendicular bisector of the line $c(A), c(B)$.

In the case that $c(A) = c(B)$, and the line $c(A), c(B)$ is required, any line which passes through $c(A)$ may be used (as a convention, we use the vertical line through $c(A)$).

(1c) Order the mappings in F as follows, dependent on whether the two subparts are to be made literal or symmetrical repetitions.

For literal repetition: f_i precedes f_j if $\theta_i^a < \theta_j^a$, where θ_k^a is the angle for mapping f_k determined as follows. Let a be the label of an arbitrary node of A . Then θ_k^a is the angle (less than 180°) between the direction of a from the center $c(A)$, and the direction of $f_k(a)$ from the center $c(B)$. If a does not determine a total

ordering on F , use another arbitrary node, b , to resolve undetermined order. Continue through the nodes until a total order is determined, resulting in $F = \{f_1, f_2, f_3, \dots, f_n\}$.

For symmetrical repetition: f_i precedes f_j if $\theta_i^a < \theta_j^a$, where θ_k^a is the angle determined as follows. Let a be the label of an arbitrary node of A . Let \bar{a}_k be the reflection of the point a with respect to the axis α_k (for the mapping f_k). Then θ_k^a is the angle (less than 180°) between the direction of \bar{a}_k from the center $c(B)$, and the direction of $f_k(a)$ from the center $c(B)$. The direction of \bar{a}_k from the center $c(B)$ may be determined by $2\alpha_k^* - a^*$, where α_k^* is the angle of α_k (either direction will do) and a^* is the angle of a from $c(A)$. If a does not determine a total ordering on F , use another arbitrary node, b , to resolve undetermined order. Continue through the nodes until a total order is determined, resulting in $F = \{f_1, f_2, f_3, \dots, f_n\}$.

Step 2 then proceeds as follows to make the two subparts either literal or symmetrical repetitions of one another:

(2) For each $f_k \in F$, in order, until a success is found, perform the appropriate step within step 2, dependent on whether literal or symmetrical repetition is desired, whether subpart A can be moved or not, and whether or not $S = \emptyset$. If no success is found for any f_k (i.e. repetition cannot be achieved), the algorithm fails. The steps

according to case are:

(2a) If only subpart B may be moved:

(i) For literal repetition where $S = \emptyset$: Based on the mapping f_k construct a layout for B which is geometrically congruent to the given layout for A , using $c(B)$ as a reference point corresponding to $c(A)$; in other words, for all $x \in A$, if $f(x) = y$, then place y so that it has the same relation to $c(B)$ as x does to $c(A)$. Succeed.

(ii) For literal repetition where $S \neq \emptyset$: determine a new center for B , $c'(B)$, using an arbitrary node $a \in S$ as follows: let $b = f_k^{-1}(a)$; then $c'(B)_x = c(A)_x - b_x + a_x$ and $c'(B)_y = c(A)_y - b_y + a_y$. Based on the mapping f_k construct a layout for B which is geometrically congruent to the given layout for A , using $c'(B)$ as a reference point corresponding to $c(A)$. If this requires moving any $c \in S$ then fail for this mapping; else, succeed.

(iii) For symmetrical repetition where $S = \emptyset$: Based on the mapping f_k , construct a layout for B which is geometrically symmetric to the given layout for A with respect to the axis α_k for the mapping f_k ; in other words, for all $x \in A$, if $f_k(x) = x'$, place x' so that the line x, x' is perpendicularly bisected by α_k . Succeed.

(iv) For symmetrical repetition where $S \neq \emptyset$: Based on the mapping f_k , construct a layout for $B' = B - S$ which is geometrically symmetric to the given layout for $A' = A - S$ with respect to the axis α_k . Succeed.

(2b) If both subparts A and B may be moved:

(i) For literal repetition where $S = \emptyset$: Construct the layout for A and B as follows. For each pair $a \in A$ and $f_k(a) = b$, we determine first an average distance, d , and then an average line segment, ℓ . From the average line segment, ℓ , we then determine new positions in the layout for a and b . To find the average distance, d , first let d_a be the distance from a to $c(A)$, and let d_b be that from b to $c(B)$. Then $d = (d_a + d_b)/2$. To then determine the average line segment, ℓ , first let b' be the point such that:

$$b'_x = b_x + c(A)_x - c(B)_x$$

$$b'_y = b_y + c(A)_y - c(B)_y$$

Then ℓ is the line segment with length d and startpoint $c(A)$ which bisects the smaller of the angles between line segments $(a, c(A))$ and $(b', c(A))$. Let us now call the other end of line segment ℓ , c . Then if $\Delta x = c_x - c(A)_x$ and $\Delta y = c_y - c(A)_y$, we determine the new positions for a and b as follows:

$$a_x = c(A)_x + \Delta x = c_x$$

$$a_y = c(A)_y + \Delta y = c_y$$

$$b_x = c(B)_x + \Delta x$$

$$b_y = c(B)_y + \Delta y$$

Succeed.

(ii) For literal repetition where $S \neq \emptyset$: Construct the layout for A and B as follows. Label each node in $A \cup B$ as p_i , $i = 1, \dots, n$, where $|A \cup B| = n$. Form $2m$ equations where $|A| = |B| = m$, as follows. For each $p_i \in A$ if $p_i = a$ and $f_k(a) = p_j$ for the equations:

$$p_{i_x} - c(A)_x = p_{j_x} - c(B)_x$$

$$p_{i_y} - c(A)_y = p_{j_y} - c(B)_y$$

We will abbreviate each pair as:

$$p_i - c(A) = p_j - c(B)$$

giving us m equations in n unknowns. The equations express the fact that each $a = p_i$ has the same x and y relationships to $c(A)$ as its image $f_k(a) = p_j$ has to $c(B)$. We then solve the equations to express all p_i 's in terms of any $n-m$ arbitrary p_i 's, say p_1, \dots, p_{n-m} . To more closely determine the value of p_1, \dots, p_{n-m} we then require that:

$$c(A)_x = (\text{Max}(p_{i_x}, p_i \in A) + \text{Min}(p_{i_x}, p_i \in A))/2$$

$$c(A)_y = (\text{Max}(p_{i_y}, p_i \in A) + \text{Min}(p_{i_y}, p_i \in A))/2$$

This assures us that the centers for A and B remain fixed. We need only check for one center, for, if these equations hold for A and not for B , it must be that for some $a \in A$, $f_k(a) \in B$, a does not

have the same x and y relationships to $c(A)$ that $f_k(a)$ has to $c(B)$. This last step may not completely determine p_1, \dots, p_{n-m} , but any assignment of values to these variables which meets the requirements of this last pair of equations is a solution and thus yields a layout. However, care must be taken so that points do not overlap.

If these last equations yield the result that two points, p_i and p_j are to be determined subject to the constraints that $p_{i_x} + p_{j_x} = k_1$ and $p_{i_y} = p_{j_y} = k_2$, one procedure for solution which seems to cause the least amount of twisting is as follows:

(a) if the line $c(A), c(B)$ tends to be more horizontal than vertical, choose $p_{i_x} = p_{j_x}$, and

(1) if $p_{i_y} \geq p_{j_y}$, choose $p_{i_y} = p_{j_y} + |c(A) - c(B)|$, and

(2) if $p_{i_y} < p_{j_y}$, choose $p_{j_y} = p_{i_y} + |c(A) - c(B)|$.

(b) if the line $c(A), c(B)$ tends to be more vertical than horizontal, choose $p_{i_y} = p_{j_y}$, and

(1) if $p_{i_x} \geq p_{j_x}$, choose $p_{i_x} = p_{j_x} + |c(A) - c(B)|$, and

(2) if $p_{i_x} < p_{j_x}$, choose $p_{j_x} = p_{i_x} + |c(A) - c(B)|$.

With p_1, \dots, p_{n-m} thus chosen, we may then determine all p_i 's from the original m equations and thus obtain a layout which

succeeds. An example of this method for finding a layout is given later in the discussion of the algorithm.

(iii) For symmetrical repetition where $S = \emptyset$: Construct the layout for A and B as follows. For each pair $a \in A$ and $f_k(a) = b$, we determine first an average distance, d , and then an average line segment, ℓ . From the average line segment, ℓ , we then determine new positions in the layout for a and b . To find the average distance, d , first let d_a be the distance from a to $c(A)$, and let d_b be that from b to $c(B)$. Then $d = (d_a + d_b)/2$. To then determine the average line segment, ℓ , first let b' be the reflection of b with respect to the axis of symmetry, α_k , for the mapping f_k . Then ℓ is the line segment with length d and startpoint $c(A)$ which bisects the smaller of the angles between line segments $(a, c(A))$ and $(b', c(B))$. Let us now call the other end of the line segment ℓ , c . The new position for a is then that of c . And the new position of b is then that of the reflection of c with respect to the axis α_k . Succeed.

(iv) For symmetrical repetition where $S \neq \emptyset$: Construct a new layout for A and B as follows. For all $a \in S$ where $f_k(a) = a$, find the point a' at which the perpendicular from a to the axis α_k intersects α_k ; then, move a to the position of a' . For all pairs $a, b \in S$ such that $f_k(a) = b$ and $f_k(b) = a$, find the node of the pair which is closest to the axis α_k , say, for example, a ; then, move a so that it lies in a position symmetric to b with respect to the axis α_k . For

all other nodes, proceed as in (2b-iii). Succeed.

We will first give some examples of the application of the algorithm, and then explain a few of the steps in more detail. Figure A2-1 shows an example of the application of each of the steps within step 2, each with a different graph. Let us now explain step 1 in more detail. In step (1a) there are several conditions under which isomorphisms are excluded. For literal repetition, condition (i) may be justified by the following two proofs:

Lemma: Let f be a mapping as specified in step (1a). Suppose that for some $a \in S$, $f(a) = a$. If A and B are made literal repetitions of one another using f , then A and B will overlap.

Proof: Since one condition for A and B to be literal repetitions is that:

$$a_x - c(A)_x = f(a)_x - c(B)_x$$

$$a_y - c(A)_y = f(a)_y - c(B)_y$$

then if $a = f(a)$, we have that $c(A)_x = c(B)_x$ and

$c(A)_y = c(B)_y$. But this implies for all pairs b , $f(b)$, that

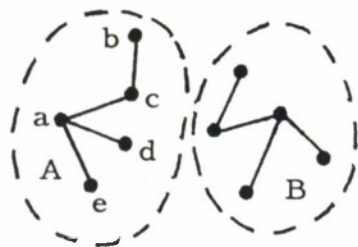
$b_x = f(b)_x$ and $b_y = f(b)_y$. Hence, A and B must overlap.

QED

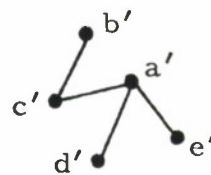
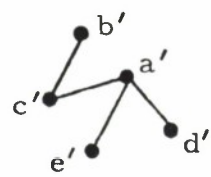
and:

original layout

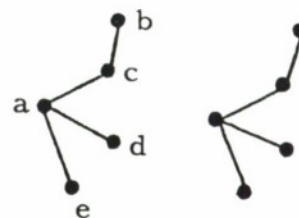
(2a-i)



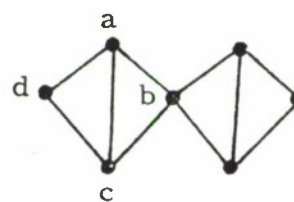
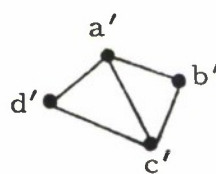
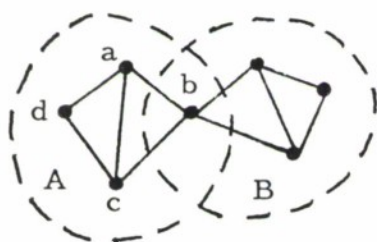
possible mappings



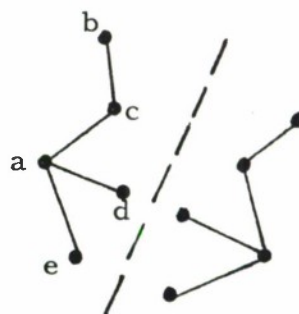
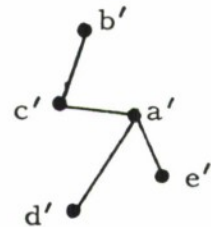
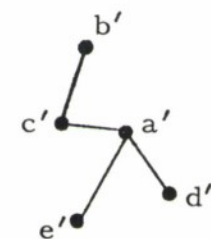
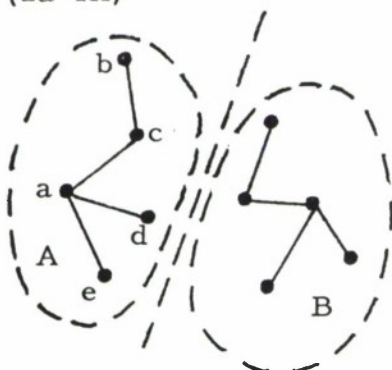
resultant layouts



(2a-ii)



(2a-iii)



(2a-iv)

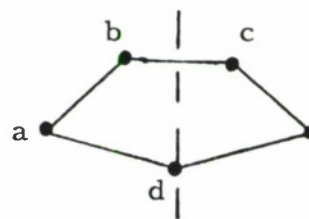
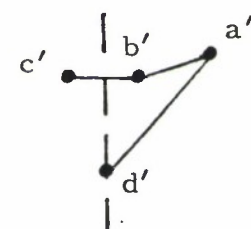
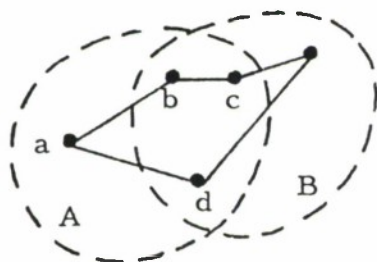
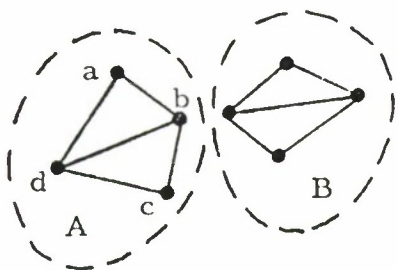
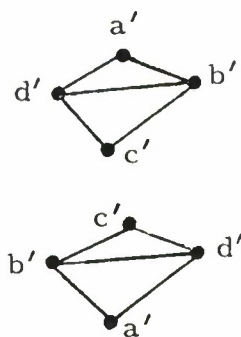


Figure A2-1

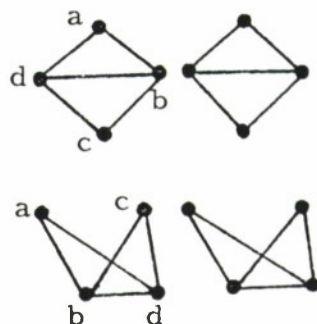
original layout
(2b-i)



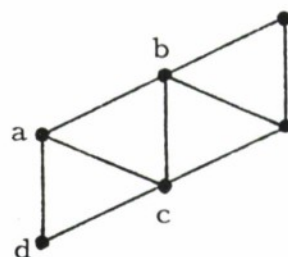
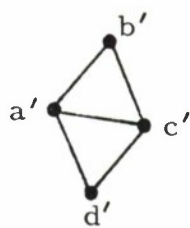
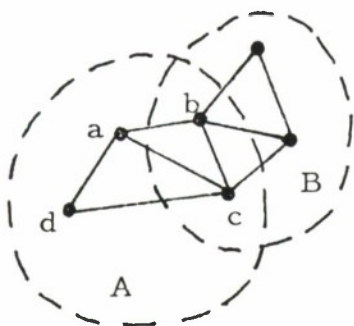
possible
mappings



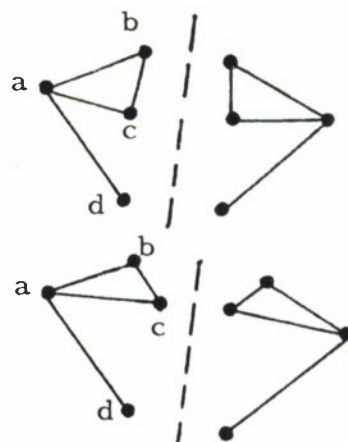
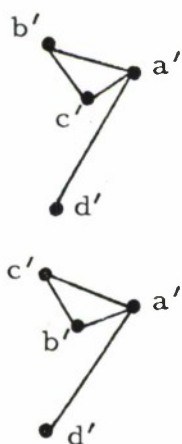
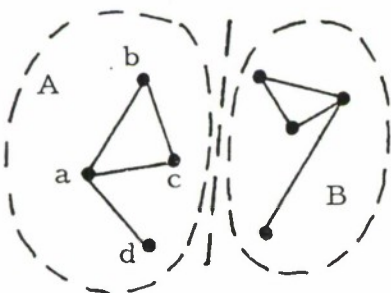
resultant
layouts



(2b-ii)



(2b-iii)



(2b-iv)

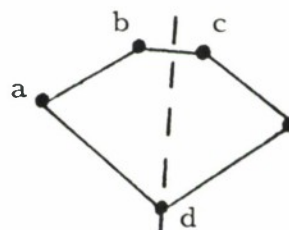
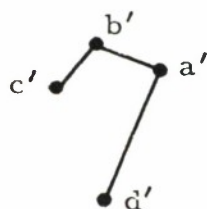
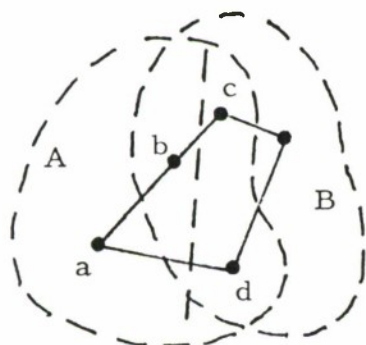


Figure A2-1 (continued)

Lemma: Let f be a mapping as specified in step (1a). Suppose that $S \neq \emptyset$, and for some $S' \neq \emptyset$, $S' \subseteq S$, that $f(S') = S'$. If A and B are made literal repetitions of one another using the map f , then A and B will overlap.

Proof: Assume for $S' \subseteq S$, $S' \neq \emptyset$, that $f(S') = S'$, and that A and B will not overlap. We will show that this leads to a contradiction. Let $|S'| = n$. Also, let the relation $a > b$ mean that node a is higher than, or, if it is at the same height, to the left of node b .

Now let a be the unique node in S' such that $a > b$ for all $b \in S'$, $b \neq a$. Since we require that no two nodes occupy the same position, a unique node a exists in S' . But since $f(a) \in S'$, then $a > f(a)$ unless $a_x = f(a)_x$ and $a_y = f(a)_y$. By our previous lemma, if $a_x = f(a)_x$ and $a_y = f(a)_y$ then A and B will overlap. Thus, it must be that $a > f(a)$.

Now by the conditions required for A and B to be literal repetitions under f , we have that:

$$a_x - c(A)_x = f(a)_x - c(B)_x$$

$$a_y - c(A)_y = f(a)_y - c(B)_y$$

and furthermore, where we use $f^2(a)$ to mean $f(f(a))$, that:

$$f(a)_x - c(A)_x = f^2(a)_x - c(B)_x$$

$$f(a)_y - c(A)_y = f^2(a)_y - c(B)_y$$

since $f(S') = S'$ and $a \in S$. But this implies that:

$$a_x - f(a)_x = f(a)_x - f^2(a)_x$$

$$a_y - f(a)_y = f(a)_y - f^2(a)_y$$

and unless we allow $f(a)_x = f^2(a)_x$ and $f(a)_y = f^2(a)_y$, which will cause A and B to overlap, $f(a) > f^2(a)$.

Similarly, we have that:

$$f^2(a) > f^3(a)$$

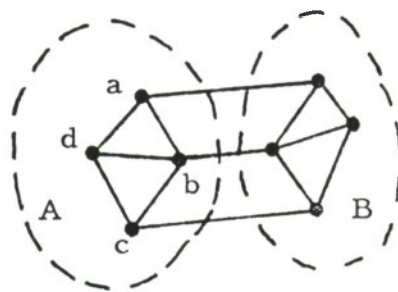
.

$$f^{n-1}(a) > f^n(a)$$

But since $|S'| = n$, $f^n(a) = f^k(a)$ for some $k < n$, and thus for some $k < n$ we have that $f^{n-1}(a) > f^k(a)$, which is a contradiction.

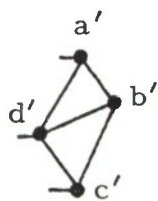
QED

Hence, removing mappings of this type avoids dead ends and overlaps. Condition (ii), the motivation for which is only intuitive at this point, is used to avoid repetition in isomorphisms, and twisting of results. For example, if we did not use this condition with the layout of figure A2-2a we would general four possible maps instead of



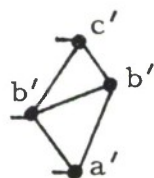
clockwise labeling

(a)

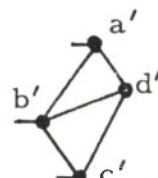


clockwise maps

(i)

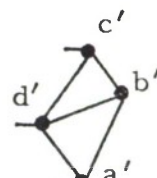


(ii)



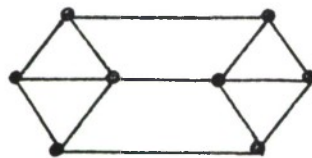
counterclockwise maps

(iii)

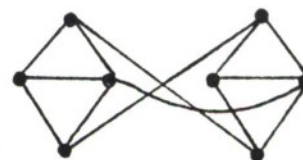


(iv)

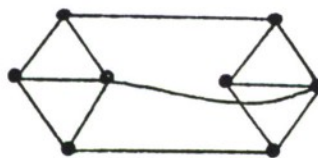
(b)



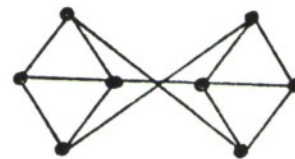
(i)



(ii)



(iii)



(iv)

(c)

Figure A2-2

two (A2-2b); the layouts resulting from these maps are shown in figure A2-2c. Further examination of condition (ii) is necessary.

The first two conditions for removal of isomorphisms in step (1a), in the case of symmetrical repetition, may also be justified by proof:

Lemma: Let f be a mapping as specified in step (1a). Suppose that for $a \in S$, $f(a) = b$ and $b \notin S$. If A and B are made symmetrical repetitions of one another using the mapping f , then A and B will overlap.

Proof: Assume $a \in S$, $f(a) = b$ and $b \notin S$, and that α is the axis of symmetry for the mapping f . Since $b \notin S$, either $b \notin A$ or $b \notin B$. Assume $b \notin A$ (the case where $b \notin B$ is similar). Then since b is to be the symmetric image of a with respect to α , the line segment (a, b) will be perpendicularly bisected by α .

Now since $a \in S$, there must be some $c \in A$ for which $f(c) = a$, according to the map formulation in step (1a). And since $b \notin A$, then $c \neq b$. But then the line segment (a, c) is also perpendicularly bisected by the axis α . Hence, b and c must have the same positions, and A and B must overlap.

QED

Lemma: Let f be a mapping as specified in step (1a). Suppose for $a, b \in S$, that $f(a) = b$ and $f(b) \neq a$. If A and B are made symmetrical repetitions of one another using the mapping f , then A and B will overlap.

Proof: Assume for $a, b \in S$ that $f(a) = b$ and $f(b) \neq a$, that that α is the axis of symmetry for the mapping f . Then $f(b) = c$ for some $c \neq a$. Now since b is to be the symmetric image of a with respect to the axis α , then the line segment (a, b) will be perpendicularly bisected by α .

Also, since c is to be the symmetric image of b with respect to the axis α , then the line segment (b, c) will also be perpendicularly bisected by α . Hence c and a must have the same positions, and A and B must overlap.

QED

Condition (iii) for symmetrical repetition is again an intuitive conjecture which must be looked into further. The reasons for its use are similar to those for condition (ii) for literal repetition.

Step (1b), the determination of the axis of symmetry for each mapping, is self-explanatory. In general, the axis is taken to be the line which separates the region of A from that of B .

Step (1c), the ordering of the isomorphisms remaining in f ,

is performed in an attempt to find solutions which involve the least amount of twisting possible. For example, in figure A2-2, map (i) is tried before map (ii), based on the directions of the node a from $c(A)$ and the node a' from $c(B)$. We see that the layout based on the mapping of (i) is less twisted in terms of the whole graph than that based on (ii). Proceeding according to step (1c) should yield a total ordering of the maps, otherwise at least two of the maps will be identical.

Step (2), which derives new layouts, is broken down into cases according to options chosen and according to whether $S = \emptyset$ or $S \neq \emptyset$. Steps (2a-i), (2a-iii), (2b-i), and (2b-iii), the cases where $S = \emptyset$ are quite straightforward. In the case where A cannot be moved, B is made to conform to A . When A can be moved, an "average" position for each node is determined.

The remainder of the steps all involve cases where $S \neq \emptyset$. These cases are more difficult to handle. In step (2a-ii), a new position for $c(B)$ must be determined which makes it possible for the nodes of S to have the correct relationships to $c(B)$. In step (2a-iv), only those nodes in A not in B need be moved to obtain symmetrical repetition; shared nodes are already symmetric. Step (2b-iv) involves moving the nodes in S in a different manner from those not in S , in order to make the shared nodes symmetric, as well as the nodes not shared.

The most complicated step, (2b-ii) requires some explanation. The first part of the step, that of labeling the nodes and forming $2m$ equations (or m abbreviated equations) is quite straightforward, as is solving for $n-m$ variables. With some of the mappings which we eliminated in step (1a), it is possible that these equations yield more than $n-m$ free variables. Furthermore, with such cases, it is also possible that the set of equations is incompatible in the following sense. Suppose we had the equation:

$$p_2 - c(A) = p_3 - c(B)$$

where p_2 and p_3 are elements of S , and where p_2 was $a \in A$ and also $f(b) \in B$ (since it is shared it has two roles), and p_3 was $b \in A$ and $f(a) \in B$. Then substituting into the equations for each subpart we obtain:

$$f(b) - f(a) = c(A) - c(B)$$

$$a - b = c(A) - c(B)$$

which unabbreviated implies:

$$a_x - b_x = c(A)_x - c(B)_x = f(b)_x - f(a)_x$$

$$a_y - b_y = c(A)_y - c(B)_y = f(b)_y - f(a)_y$$

This condition is incompatible since a must relate to b in the same way $f(a)$ relates to $f(b)$.

However, we conjecture that such cases do not arise once

step (1a) is performed (step (1a) eliminates the example above since $f(S') = S'$ for $S' \subseteq S$. Thus we do not include checks for number of unknowns and for compatibility in step (2b-ii). However, this conjecture should be looked into further.

As noted in the description, the condition that centers remain centers in (2b-ii) may not completely determine the unknowns. When arbitrary values may be given to the unknowns it is wise to avoid changing the order of placement along the x or y axes where possible, as such a change may again cause twisting. This may be seen in the last step of the example below.

To clarify the procedure of step (2b-ii), we will show how it applies to the layout and mapping of figure A2-3, with node labeling as shown.

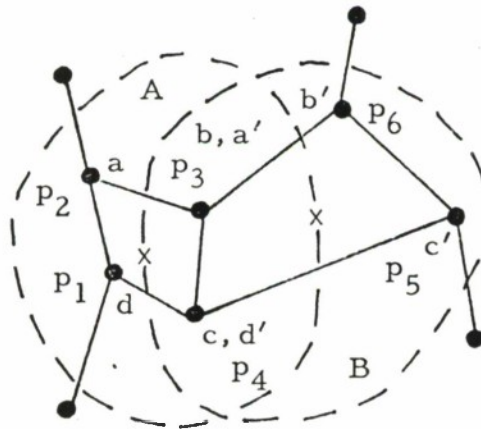


Figure A2-3

The four (=m) equations in six (=n) unknowns for this example are:

$$p_1 - c(A) = p_4 - c(B)$$

$$p_2 - c(A) = p_3 - c(B)$$

$$p_3 - c(A) = p_6 - c(B)$$

$$p_4 - c(A) = p_5 - c(B)$$

The equations solved in terms of two (=n-m) unknowns are:

$$p_1 = p_5 - 2c(B) + 2c(A)$$

$$p_4 = p_5 - c(B) + c(A)$$

$$p_3 = p_6 - c(B) + c(A)$$

$$p_2 = p_6 - 2c(B) + 2c(A)$$

Using $c(A) = (1, 2)$ and $c(B) = (3, 3)$ they become:

$$p_{1_x} = p_{5_x} - 4 \quad p_{1_y} = p_{5_y} - 2$$

$$p_{4_x} = p_{5_x} - 2 \quad p_{4_y} = p_{5_y} - 1$$

$$p_{3_x} = p_{6_x} - 2 \quad p_{3_y} = p_{6_y} - 1$$

$$p_{2_x} = p_{6_x} - 4 \quad p_{2_y} = p_{6_y} - 2$$

Applying the center criterion we have:

$$(\text{Max}(p_{5_x} - 4, p_{6_x} - 4, p_{6_x} - 2, p_{5_x} - 2) + \text{Min}(p_{5_x} - 4, p_{6_x} - 4, p_{6_x} - 2, p_{5_x} - 2)) / 2 = 1$$

$$p_{5_x} + p_{6_x} - 6 = 2$$

$$p_{5_x} + p_{6_x} = 8$$

$$(\text{Max}(p_{5y} - 2, p_{6y} - 2, p_{6y} - 1, p_{5y} - 1) + \text{Min}(p_{5y} - 2, p_{6y} - 2, p_{6y} - 1, p_{5y} - 1)) / 2 = 2$$

$$p_{5y} + p_{6y} - 3 = 4$$

$$p_{5y} + p_{6y} = 7$$

Using the method suggested in the algorithm for choosing values

for $p_{5_{x,y}}$ and $p_{6_{x,y}}$ we obtain:

$$p_{5x} = p_{6x} = 4$$

and since $p_{6y} > p_{5y}$ in figure A2-3

$$p_{6y} = p_{5y} + 2$$

and thus $p_{6y} = 9/2$ and $p_{5y} = 5/2$. We then have that:

$$p_1 = (0, 1/2)$$

$$p_2 = (0, 5/2)$$

$$p_3 = (2, 7/2)$$

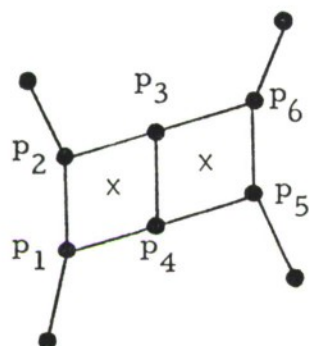
$$p_4 = (2, 3/2)$$

$$p_5 = (4, 5/2)$$

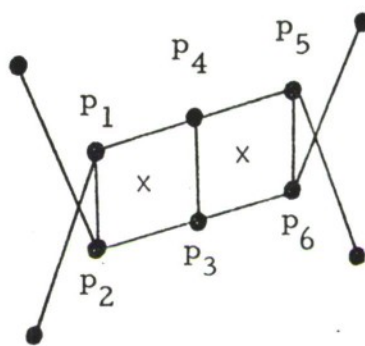
$$p_6 = (4, 9/2)$$

as can be seen in figure A2-4a. If we had ignored the fact that

$p_{6y} > p_{5y}$, we might let $p_{6y} + 2 = p_{5y}$, and thus obtain the twisted figure in A2-4b, as was anticipated above.



(a)



(b)

Figure A2-4

Appendix 3

AN ALGORITHM FOR LINK LENGTH CONSISTENCY

An algorithm to improve link length consistency, as measured by the number of different lengths appearing in a layout is described here. In the description, bend points are treated as nodes of degree two, and the word link should be taken to mean link segment.

As described briefly in section 2.2.2.3, this algorithm is partly constructive, although it uses some of the original layout, and so is classified as a modifying algorithm. It proceeds by breaking the layout up into cycles of minimal length, and generating regular polygons for these cycles wherever possible.

In more detail, the algorithm first removes all dangling trees from the layout (steps 1-3), one link at a time, recording the order in which links are removed. After this is done, the remainder of the layout is broken up into non-separating components (in the graph theoretic sense) (step 5).

Each of these components is then processed one at a time (steps 6-33), in order to derive layouts for the components. The order in which the components are processed is arbitrary, except that there is a preference for processing those components first which have nodes in common with already processed components.

The processing of a component begins, again by the removal

and recording of all dangling trees from the component (steps 7-9), one link at a time. What remains are the cycles of the component.

We begin processing the remainder of the component by finding the cycle of minimal length, and forming a regular polygon of its sides (steps 6-17), where the sides are a length which has been used before (perhaps in another component which has already been processed). If this is the first component to be processed, we will, of course, determine the first length to be used in the layout here. The first cycle has thus been placed.

We next look for other cycles of the component which are attached to those already placed, and which have not yet been placed. We first look for a smallest cycle which shares links with already placed cycles, and try to place it (steps 17-30) by forming a regular polygon of its unplaced links with sides which are the same length as the distance between the two points at which it is attached to already placed cycles. If, with this placement we cause node or link overlap, we throw away this placement, and try another cycle. This series of steps is then repeated.

At some point (step 31) we have either placed all the cycles of the component, or found that they cannot be placed without causing overlap. If the latter is the case, we throw away any previous placement in the component (step 33) and layout the whole cyclic part of the component as a single regular polygon. If we have found a placement

for all cycles, but the number of lengths resulting exceeds the number we would obtain by forming a single regular polygon (step 32), we again throw away the placement and form the single polygon. Otherwise, the placement is kept.

The placement of the cyclic portion of the component is then complete, and the dangling trees of the component are restored in the order in which they were removed (steps 11-13) so as not to increase the total number of lengths in the layout. The processing of this component is then completed (step 14) by placing it correctly in the total layout, relative to other already processed components. We then go on to process another component.

When all components have been processed, we restore the dangling trees initially removed from the layout in the order removed, in such a way as not to increase the total number of lengths in the layout (step 34-35) and terminate the algorithm.

A few preliminaries should be noted before we state the algorithm:

LEN is a set of lengths.

PLACED is a set of nodes.

REM is a set of triples.

TEMPLLEN is a set of lengths.

TEMPREM is a set of triples.

OLD(z) is the pair of x, y -coordinates of node z in the original layout.

NEW(z) is the pair of x, y -coordinates of node z in the resultant layout.

CYCLES is a set of cycles in the layout, each recorded as a sequence of nodes.

The algorithm is then:

1) Set $l \Rightarrow K$, $REM = \emptyset$, $PLACED = \emptyset$, $LEN = \emptyset$. For each node x , set OLD(x) \Rightarrow NEW(x). Go to step 2.

2) If any nodes are of degree one, go to step 3; else, go to step 4.

3) Remove those nodes presently of degree one from the layout. For each node removed, add a triple to REM of the form: (removed node, adjacent node, K). Add one to K , and go to step 2.

4) If only nodes of degree zero remain in the layout, go to step 34; else, go to step 5.

5) Break the layout into non-separating components, C_1, \dots, C_n . This is done as follows: for each node, in turn, separate the node into n nodes if the node is of degree n . In each resulting distinct part of the layout, if there are two or more nodes resulting from the original node in the same connected part, join them back into one. Then try this for the next node on each part of the layout thus formed. Nodes of degree

zero are discarded. An example of this process is shown in figure A3-1. Each component, C_i , is considered unprocessed. Go to step 6.

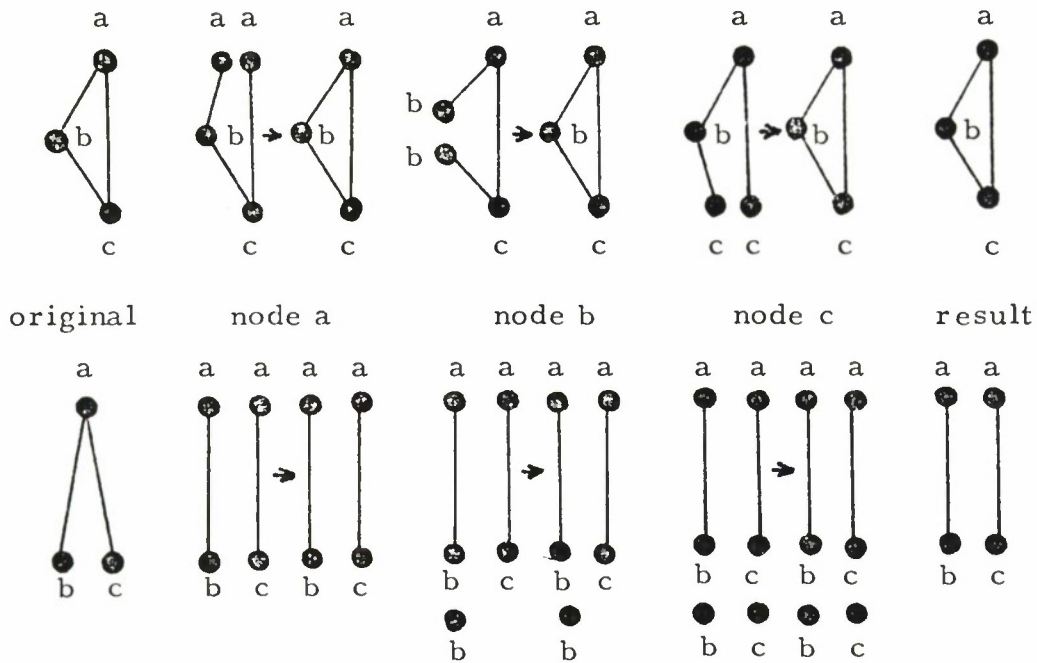


Figure A3-1

6) If all components C_1, \dots, C_n have been processed, go to step 34. Else, if $PLACED = \emptyset$, set $z = 0$, choose any component C_i as the current component, C , and go to step 7. If $PLACED \neq \emptyset$ look for an unprocessed component C_i which contains a node x such that $x \in PLACED$. Make this component the current one, C , set $x \Rightarrow z$ and $NEW(x) \Rightarrow RNEW$, and go to step 7. If $PLACED \neq \emptyset$, but there is no unprocessed component C_i with $x \in C_i$ and $x \in PLACED$, then choose any unprocessed component as C , set $z = 0$, and go to step 7.

7) Set $1 \Rightarrow J$, $TEMPREM = \emptyset$, $TEMPLLEN = \emptyset$, and go to step 8.

8) If there are nodes of degree one in C, go to step 9; else, go to step 10.

9) Except for the node z, remove those nodes presently of degree one from the layout. For each node removed, add a triple to TEMPREM of the form: (removed node, adjacent node, J). Add one to J, and go to step 8.

10) If only nodes of degree zero remain in C, go to step 11; else, go to step 15.

11) Set L equal to the average of the lengths of the links originally in C. If LENUTEMPLLEN = \emptyset , set $L \Rightarrow M$, add L to TEMPLLEN, and go to step 12. If LENUTEMPLLEN $\neq \emptyset$, set M to that element of LENUTEMPLLEN which is closest to L, and go to step 12.

12) Subtract one from J, and go to step 13.

13) If $J = 0$, go to step 14. If $J \neq 0$, find all triples in TEMPREM of the form: (node1, node2, J). For each such triple, in turn, set NEW(node1) as follows: if $OLD(node2) = x_2, y_2$, $OLD(node1) = x_1, y_1$, and $NEW(node2) = x', y'$, then $NEW(node1) = x, y$ where:

$$\frac{x_2 - x_1}{y_2 - y_1} = \frac{x' - x}{y' - y}, \text{ sign } (x_2 - x_1) = \text{sign } (x' - x),$$

and
$$\sqrt{(x - x')^2 + (y - y')^2} = M$$

Go to step 12.

(14) If $z \neq 0$ calculate:

$$\Delta x = x' - x''$$

$$\Delta y = y' - y''$$

where $RNEW = x', y'$, $NEW(z) = x'', y''$, and for each node a , $a \in C$, where $NEW(a) = x, y$, set $NEW(a) = x + \Delta x, y + \Delta y$. In any case, add the nodes of C to $PLACED$ and the lengths of $TEMPLN$ to LEN . Component C has thus been processed; go to step 6.

15) Call the remainder of component C , C' . All link segments in C' are now considered to be unmarked, and all nodes to be unplaced. Look for a simple cycle (no node appears twice except for the first and last) of minimal length in C' . If there is a choice of minimal cycles, and if $z \neq 0$, choose one which doesn't contain z , if possible. Call the chosen cycle p . Set L equal to the average of the lengths of the links of the cycle p in the original layout (i.e. use $OLD(x)$). Set c_x, c_y as the center of the original positions of the nodes in the cycle p . If $LEN = \emptyset$, set $L \Rightarrow M$, and go to step 16. If $LEN \neq \emptyset$, set M to that element of LEN which is closest to L , and go to step 16.

16) Add M to $TEMPLN$. Form a regular polygon from the cycle p , with center c_x, c_y , and sides of length M . Choose $a \in p'$ so that if $z \in p$, then $z \Rightarrow a$, and if $z \notin p$, then $x \Rightarrow a$ for some arbitrary $x \in p$. Then orient the regular polygon so that the direction of the new position of a from c_x, c_y is the same as the direction of

OLD(a) from c_x, c_y . Record the new positions of all $x \in p$ as NEW(x). Mark each link in the cycle p as fixed and mark each node in the cycle as placed. Go to step 17.

17) If the number of unplaced nodes in C' is zero, go to step 32; else, set $0 \Rightarrow \text{UNF}$, $1 \Rightarrow \text{FX}$, and go to step 21.

18) Add one to UNF. If UNF is greater than the number of links marked unfixed in C' , then go to step 31; else, go to step 19.

19) Set $0 \Rightarrow \text{FX}$ and go to step 21.

20) Add one to FX. If FX is greater than the number of links marked fixed in C' , then go to step 18; else, go to step 21.

21) Set $0 \Rightarrow N$; go to step 22.

22) Add one to N, and go to step 23.

23) If N is larger than the number of unplaced nodes in C' , then go to step 20; else, go to step 24.

24) Set $\text{CYCLES} = \emptyset$. Look for the set of simple cycles of length $N + \text{FX} + \text{UNF} + 1$ (the length of a cycle is the number of nodes it includes) with FX links marked fixed, UNF links marked unfixed, and $N+1$ links unmarked, containing N unplaced nodes. The unplaced nodes must be consecutive in the cycle. Set CYCLES equal to this set, and go to step 25.

25) If $\text{CYCLES} = \emptyset$, go to step 22; else, go to step 26.

26) Choose any cycle q in CYCLES, and remove it from CYCLES, if possible choosing q so that if $z \neq 0$, $z \notin q$. Determine

the two points of contact of the unplaced part of the cycle with the placed part. These two points are the placed nodes a and b which surround the sequence of unplaced nodes in q . Draw an imaginary line, a, b on the layout and count the placed nodes on either side of the line. Call the side with fewer nodes S_1 and the other S_2 . If both sides have the same number of nodes, and if z is placed on either side, call the side not containing z , S_1 . Set $S_1 \Rightarrow S$, and go to step 27.

27) Position the unplaced nodes in q , say n_1, \dots, n_n , in order, on side S of line a, b , so that a regular polygon of $n+2$ sides is formed with the line segment a, b as one side. For example, if $q = (a, n_1, n_2, n_3, b)$, we would have the positioning in figure A3-2.

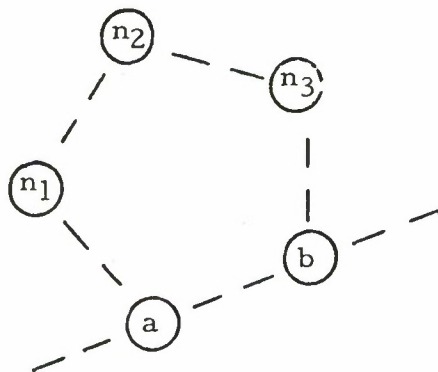


Figure A3-2

28) Draw all links of the layout connecting the nodes n_1, \dots, n_n with each other and with those nodes already marked as placed. Check that neither of the following conditions exists:

- a) a node lies on top of another node or link.
- b) a link goes through a node to which it is not attached.

If neither of these conditions exists, go to step 30; otherwise, if $S = S_1$, go to step 29; else, go to step 25.

29) Set $S_2 \Rightarrow S$, and go to step 27.

30) Record the positions of n_1, \dots, n_n as $NEW(n_1), \dots, NEW(n_n)$ and mark these nodes as placed. Mark all fixed links of the cycle q as unfixed. Mark all other links of q as fixed. Mark all links not in q but attached to an $n_i \in q$ ($i = 1, \dots, n$) as unfixed and record their lengths in $TEMPLEN$ as well as the length a, b . Go to step 17.

31) If not all nodes of C' are placed, go to step 33; else, to step 32.

32) Set $N_n = \left\{ \begin{array}{ll} 1 + (n-3)/2 & n \text{ odd} \\ 1 + (n-2)/2 & n \text{ even} \end{array} \right\}$ where n is the number of nodes in C' . If $N < |TEMPLEN|$ go to step 33; else, go to step 11.

33) Find the center point of the old positioning of the nodes of C' , and call it p_x, p_y . Form a regular polygon of the n nodes in C' , the center of which lies at p_x, p_y . The length M of the sides of the polygon will be that element of LEN which is closest to the average of the link lengths originally in C' , or, if $LEN = \emptyset$, that average length. The arrangement of nodes along the sides of the polygon should be as close as possible to their original arrangement. Record the resultant position for each node $x \in C'$ as $NEW(x)$. Draw all links in C' , and record all lengths in $TEMPLEN$. Go to step 11.

34) Set $K-1 \Rightarrow K$; go to step 35.

35) If $K=0$, exit (positions in the new layout are those given by $NEW(x)$; the number of lengths is $|LEN|$). If $K \neq 0$, find all triples in REM of the form: $(node1, node2, K)$. For each such triple, in turn, set $NEW(node1)$ as follows: if $OLD(node1) = x_1, y_1$, $OLD(node2) = x_2, y_2$, and $NEW(node2) = x', y'$, then $NEW(node1) = x, y$ where:

$$\frac{x_2 - x_1}{y_2 - y_1} = \frac{x' - x}{y' - y}, \text{ sign}(x_2 - x_1) = \text{sign}(x' - x),$$

and $\sqrt{(x-x')^2 + (y-y')^2}$ equals that element of LEN which is closest in length to the distance between $OLD(node1)$ and $OLD(node2)$. If, when this triple is processed, $LEN = \emptyset$, use the distance between $OLD(node1)$ and $OLD(node2)$ and add it to LEN . Go to step 34.

An example of the application of this algorithm will be helpful. Suppose we have the layout of figure A3-3a. Applying steps 1-3, we obtain figure A3-3b, which is broken into components C_1, \dots, C_5 in step 5 as shown in A3-3c. In step 6 we choose a first component to process, say C_1 . Processing C_1 we find no change from steps 7-10. In step 15, we then choose a minimal cycle, say $(5, 6, 7)$, and draw it as a regular polygon in steps 15-16. Proceeding through steps 17-30, we choose (with $UNF=0$, $FX=1$, $N=1$) the cycle $(5, 6, 3)$ and install it. We then install cycle $(7, 8, 4, 5)$ since the next group of cycles that may be installed without violating the conditions of step 28 requires that

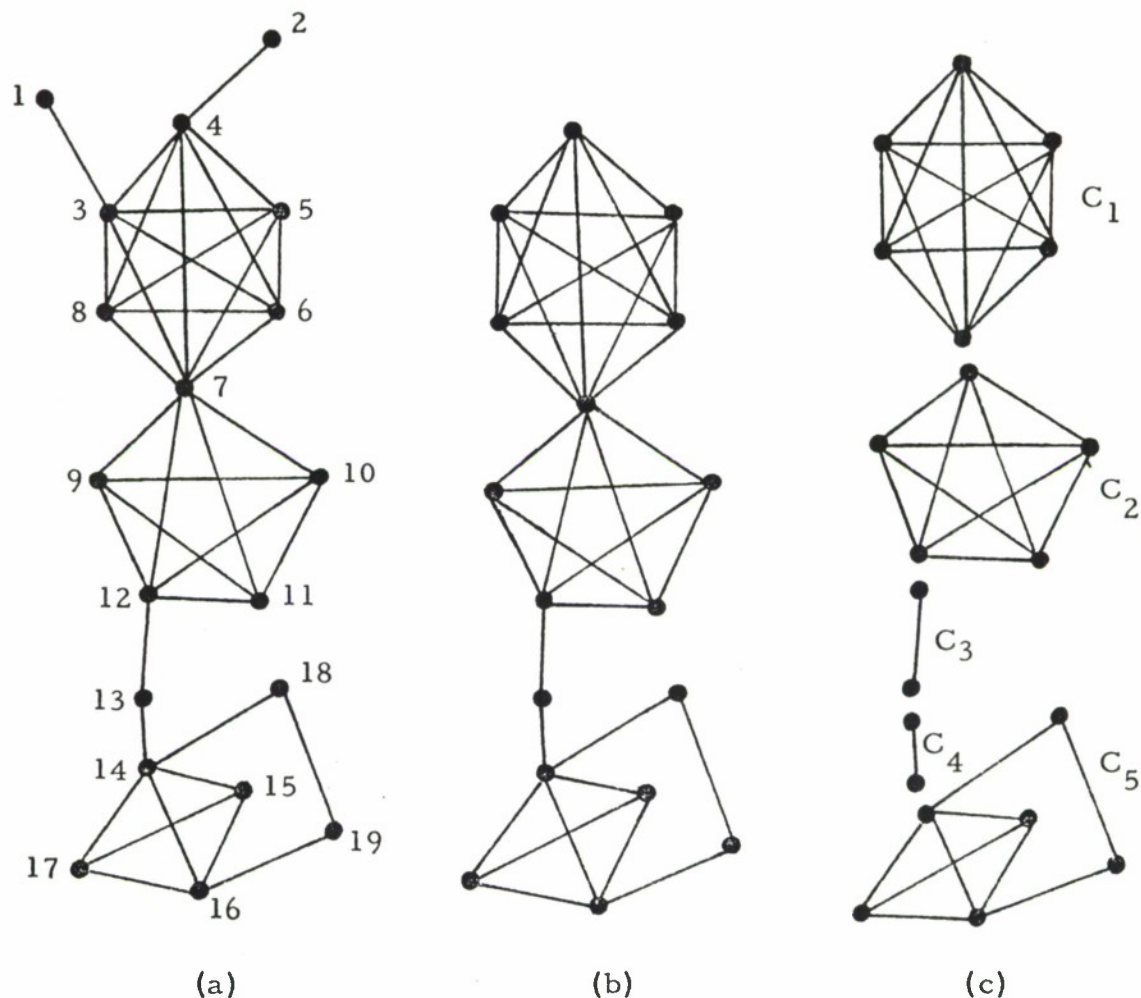


Figure A3-3

UNF = 0, FX = 1, and N = 2. We have thus placed all nodes in C_1 , as shown in figure A3-4a, using five different lengths. But checking the maximum number of lengths allowed for a layout of six nodes in step 32, we find that $N_6 = 3$, and, proceeding through step 33, our result will be the layout of figure A3-4b, with three different lengths.

Returning to step 6, we choose our next component so that it contains a node already placed. Thus, we choose C_2 , with $z = 7$.

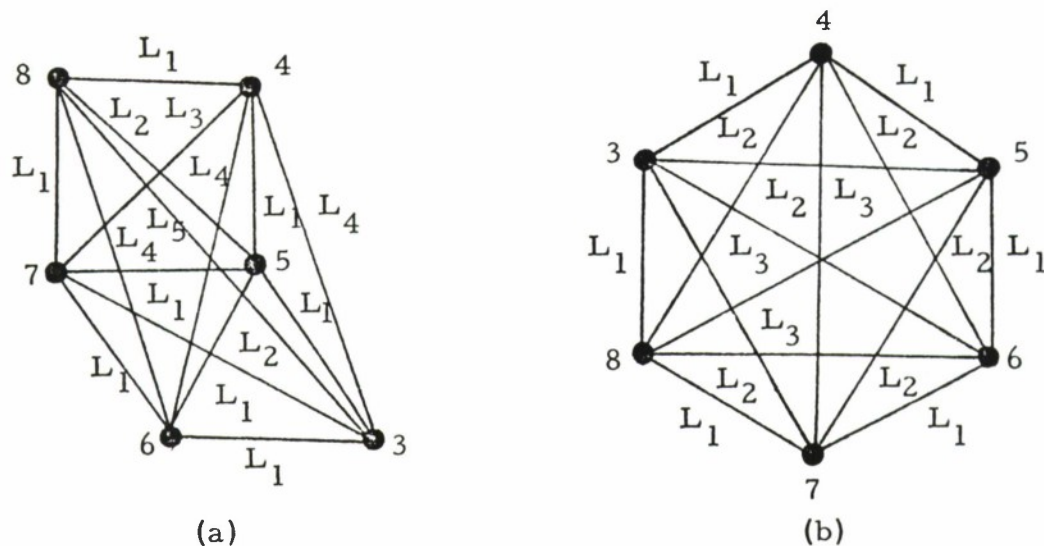


Figure A3-4

Proceeding again as for C_1 , we find that we can go as far as the placement in figure A3-5a, but cannot place the last node without violating step 28. Thus we proceed through step 31 to step 33 to obtain the placement in figure A3-5b, with two lengths, one of which appears in C_1 . Proceeding back to steps 11-14, we obtain an adjustment for the node positions of C_2 , so that the placement of the node in common with C_1 , node 7, corresponds to its previous positioning.

Again at step 6, we choose component C_3 with $z = 12$. After removing node 13 in step 9, we find that step 10 sends us to steps 11-14, where we replace node 13 so that the length corresponds to one used in $C_1 \cup C_2$, and the placement of the component C_3 fits with that of $C_1 \cup C_2$. Similarly, C_4 is chosen next with $z = 13$, and processed in a manner analogous to C_3 .

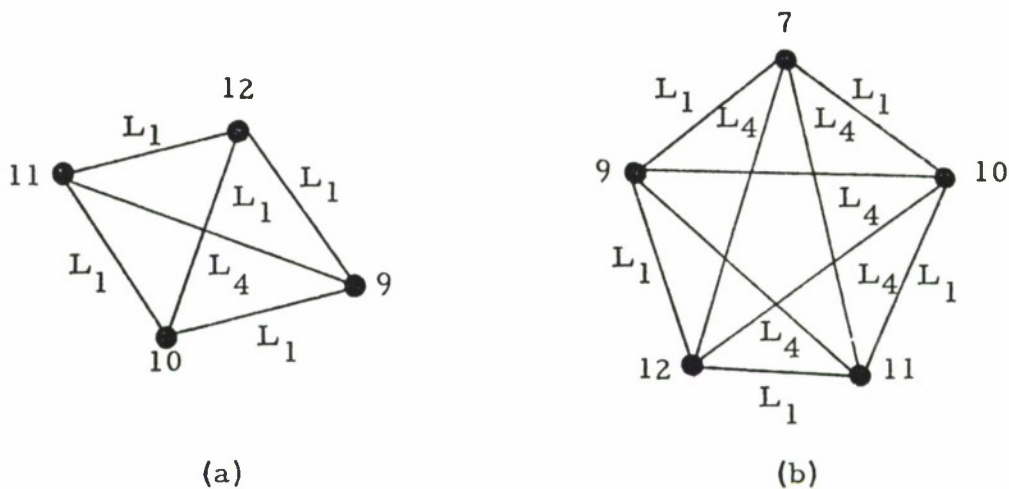


Figure A3-5

Finally we process C_5 with $z = 14$. The placement shown in figure A3-6 is obtained by choosing the cycle (14, 15, 16) in step 15, and cycles (14, 17, 16) with $UNF = 0$, $FX = 1$, and $N = 1$, and (14, 18, 19, 16, 15) with $UNF = 0$, $FX = 2$, $N = 2$. The number of lengths is two, which is less than $N_6 = 3$, and so the layout of figure A3-6 is retained for C_5 .

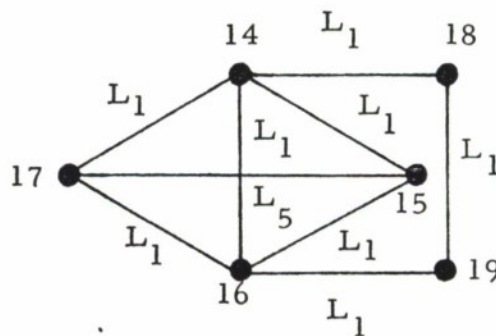


Figure A3-6

Adjusting the node positions of C_5 to fit with those of C_1-C_4 we obtain the layout of figure A3-7a. Steps 34-35 then replace nodes 1

and 2 as shown in A3-7b, so that the total number of lengths is five for the layout. The layout is then complete, and, in fact,

$$|\text{LEN}| = 5 < 9 = N_{19}.$$

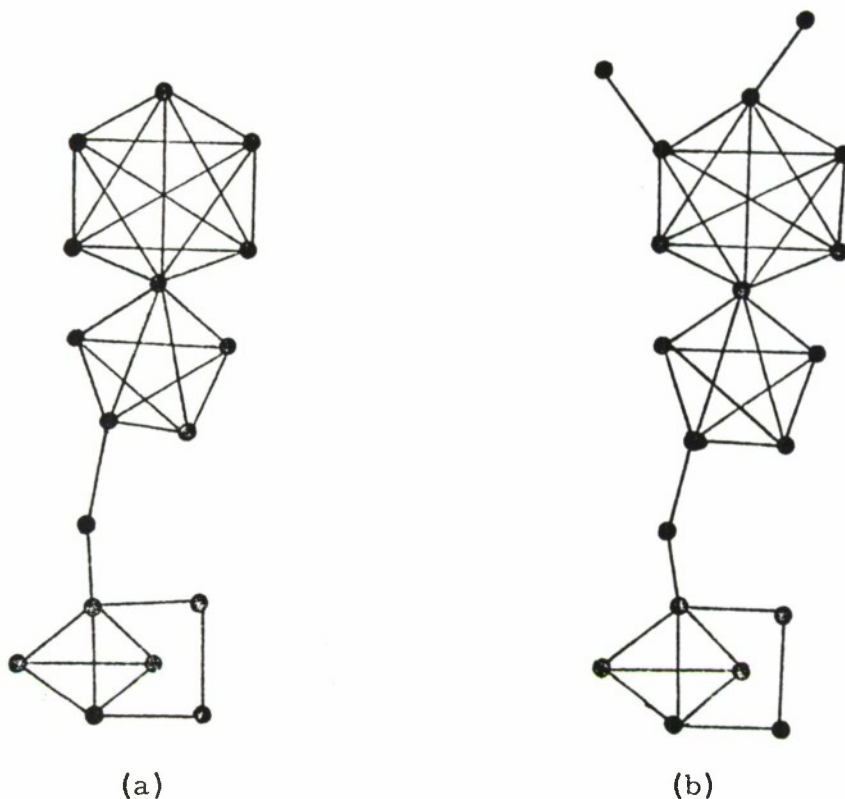


Figure A3-7

There are several ways in which this algorithm may be modified. However, at the present time, no experimentation has been performed to determine whether or not these changes actually should be made. The most important area of question is in steps 17-33. It would be more desirable, if, in stepping through the loops in 17-30, conditions leading to the situations discovered in 31 and 32 could be determined earlier, and the procedure of the algorithm corrected accordingly. In

particular, perhaps the algorithm should be aware of maximal cycles as well as minimal ones. There is also some question as to the effect of the ordering of cycle examination in these steps. As it now stands, the algorithm may find some larger cycles before finding smaller ones, since all fixed link cycles are examined first. Whether or not this makes a difference is not known.

Even with the algorithm as stated, several problems will be encountered in its implementation. Generation of regular polygons will be complicated. Generation of cycles for a given UNF, FX, and N is not a simple task either, although an algorithm has been written by Tiernan (36) which generates the simple cycles of a given graph. Finally, the check required in step 28 will be quite time-consuming to perform on a computer. It would be quite an achievement to produce some method of placement which would guarantee no violation of the conditions of this step. Thus, there is much room for work on this algorithm, but it is felt that the basic idea is a viable approach to the problem.

In order to show that we can guarantee that the layout resulting from the process just described contains N_n or fewer lengths, given a layout with n nodes, we have the following proof:

Lemma: Let G be a layout with n nodes and at least one link (hence, at least two nodes), then the layout resulting from the above algorithm will contain N_n or fewer lengths, where:

$$N_n = \begin{cases} 1 + (n-3)/2 & n \text{ odd} \\ 1 + (n-2)/2 & n \text{ even} \end{cases}$$

Proof: We first consider the results of steps 3-4; there are two cases:

a) If steps 3-4 remove all links of G the resultant layout will have exactly one length in which case the number of lengths is less than or equal to N_n .

b) If steps 3-4 do not remove all links of G , then replacing those removed in steps 34-35 adds no new link lengths. We must then be sure that for the links and nodes not removed in steps 3-4, the number of lengths in the resultant layout is less than N_n . We do this as follows. Let G' be the layout after steps 3-4 have been performed. Let p be the number of nodes in G' . Since $p \leq n$, certainly, $N_p \leq N_n$. Now we must show that if we break the p node layout, G' , up into c components as in the algorithm, where for the i th component, the number of nodes is p_i , and where:

$$\sum_{i=1}^c p_i - (c-1) = p$$

(since shared nodes should only be counted once), that:

$$N_p \geq 1 + \sum_{i=1}^c (N_{p_i} - 1)$$

For, the right hand side of this inequality is the maximal number of link lengths we can obtain, processing each component separately, according to the algorithm. This is because for the first component processed with, say, m nodes, we guarantee no more than N_m lengths. And, for any other components, say, with q nodes, since at least one link length is chosen from the group of lengths already used, we can guarantee no more than $N_q - 1$ additional lengths. The removal and subsequent replacement of nodes of degree one, in step 8-9 and steps 12-13, adds no additional lengths except for the case where nothing remains in a component after steps 7-8. In this case if this is the first component, one length is added as expected, and, if it is not the first, no new lengths are added.

Thus it remains to be shown for:

$$\sum_{i=1}^c p_i - (c-1) = p$$

that:

$$N_p \geq 1 + \sum_{i=1}^c (N_{p_i} - 1)$$

We do this by induction on c the number of components.

1) For $c = 1$ we have:

$$N_p \geq 1 + N_{p_1} - 1 = N_{p_1}$$

where $p_1 - (1-1) = p_1 = p$ and thus

$$N_p \geq N_p.$$

2) For $c = 2$ we have two cases to consider; let

p_1 and p_2 be the number of nodes in the two components, respectively, where $p_1 + p_2 = (2-1) = p_1 + p_2 - 1 = p$. Then:

i) If p is odd, then $N_p = 1 + (p-3)/2$. Since $c = 2$ we must have either that both components contain an even number of nodes, or both contain an odd number. In the first case:

$$\begin{aligned} 1 + \sum_{i=1}^2 (N_{p_i} - 1) &= 1 + N_{p_1} - 1 + N_{p_2} - 1 \\ &= 1 + (p_1 - 2)/2 - 1 + 1 + (p_2 - 2)/2 - 1 + 1 \\ &= 1 + (p_1 + p_2 - 4)/2 \\ &= 1 + (p - 3)/2 \leq N_p \end{aligned}$$

In the second case:

$$\begin{aligned} 1 + \sum_{i=1}^2 (N_{p_i} - 1) &= 1 + N_{p_1} - 1 + N_{p_2} - 1 \\ &= 1 + (p_1 - 3)/2 - 1 + 1 + (p_2 - 3)/2 - 1 + 1 \end{aligned}$$

$$= 1 + (p_1 + p_2 - 6)/2$$

$$= 1 + (p-5)/2 \leq N_p$$

ii) If p is even, then $N_p = 1 + (p-2)/2$. Since $c=2$ we must have that one component has an even number of nodes (say, p_1) and one has an odd number (say, p_2). Then:

$$\begin{aligned} 1 + \sum_{i=1}^2 (N_{p_i} - 1) &= 1 + N_{p_1} - 1 + N_{p_2} - 1 \\ &= 1 + (p_1 - 2)/2 - 1 + 1 + (p_2 - 3)/2 - 1 + 1 \\ &= 1 + (p_1 + p_2 - 5)/2 \\ &= 1 + (p-4)/2 \leq N_p \end{aligned}$$

3) Assume for $c=m$ that the inequality holds.

4) For $c=m+1$, let G' be a graph with p nodes and $m+1$ components. Form the layout G'' with m components by adding an extra link between two adjacent components k_1 and k_2 to form the single component k . The new link should not involve the node shared between k_1 and k_2 . (This can always be done since components have at least two nodes and share at most one). An example of the modification is shown in figure A3-8. Then for the new graph G'' we have

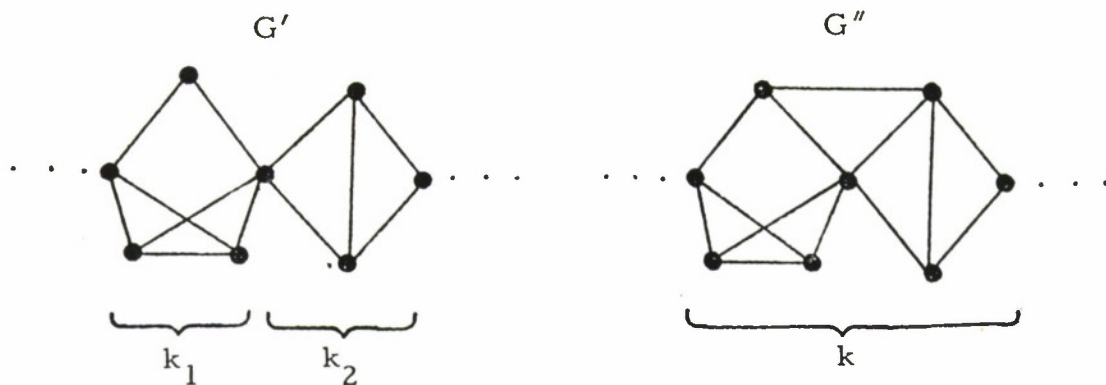


Figure A3-8

that:

$$N_p \geq 1 + \sum_{i=1}^m (N_{p_i} - 1)$$

by the induction hypothesis. But for G' , we have that:

$$1 + \sum_{i=1}^{m+1} (N_{p_i} - 1) = 1 + \sum_{i=1}^m (N_{p_i} - 1) - (N_{p_k} - 1) + (N_{p_{k_1}} - 1) + (N_{p_{k_2}} - 1)$$

where $p_k = p_{k_1} + p_{k_2} - 1$, since unchanged components give the same contribution, and the contributions of components k_1 and k_2 replace that of component k . But by part 2, for:

$$\sum_{i=1}^2 p_i - (2-1) = \sum_{i=1}^2 p_i - 1 = p$$

we have that:

$$N_p \geq 1 + \sum_{i=1}^2 (N_{p_i} - 1)$$

or that:

$$(N_p - 1) \geq (N_{p_1} - 1) + (N_{p_2} - 1)$$

But since $p_k = p_{k_1} + p_{k_2} - 1$, then:

$$(N_{p_k} - 1) \geq (N_{p_{k_1}} - 1) + (N_{p_{k_2}} - 1)$$

and thus:

$$1 + \sum_{i=1}^{m+1} (N_{p_i} - 1) \leq 1 + \sum_{i=1}^m (N_{p_i} - 1) .$$

QED

Appendix 4

AN ALGORITHM FOR PARALLELISM

This algorithm applies to a layout to increase the amount of parallelism (or to decrease the number of link slopes) in the link segments of the layout. The first part of the algorithm (steps 1-5) places the link segments into sets, called S-sets according to the following requirements:

- 1) Every link segment is in one and only one S-set,
- 2) The first element placed in an S-set is called the key link segment of that S-set,
- 3) Each S-set contains at most one subset of parallel link segments, and, if it contains one such subset, the subset includes the key link segment of the S-set,
- 4) Any link segment in an S-set has a slope within r degrees of the slope of the key link segment of the S-set, where r is determined by the user, and
- 5) Any link segment in an S-set which is not parallel to the key link segment of the S-set, is not parallel to any other link segment in the layout.

In addition, in the first part of the algorithm, once S-sets are formed, a c-value or constraint-value is assigned to each node and bend point in the layout. This value is used later in the algorithm to indicate

whether or not, in order to increase parallelism, these points may be moved, and if so, in what manner. When the c -value of a point is zero, its movement is unrestricted. When the c -value is one, the point may only be moved along a specific line. And when the c -value is two or greater, the point may not be moved. The reason for these constraints on points is that once link segments are parallel, or are made parallel, we want to maintain this parallelism.

The c -values are assigned as follows: Initially the c -value for a point, p , is zero. Then for each S -set, S , we increase the c -value of p by one if both of the conditions below are met:

- 1) p is the endpoint of some link segment, ℓ in S , and
- 2) ℓ has parallels in S .

As the algorithm progresses, and more link segments are made parallel, c -values are changed appropriately.

The second part of the algorithm (steps 6-18) consists of processing each S -set, in turn, starting with those containing the largest number of link segments. The aim of processing an S -set is to adjust the link segments of the set so that they are all parallel. We need not, of course, process S -sets containing only one element.

The first part of processing an S -set (steps 6-10) is to determine what the resultant slope of all elements in the S -set will be. This is done by choosing an element of the S -set to whose slope all other elements of the S -set will be made to conform. This selection is aimed at

maximizing the number of elements in the S-set which may be made parallel, considering the current amount of parallelism and the c-values of the elements in the S-set. Once the element with the desired slope is so determined, it is made the new key link segment of the S-set. At this point all elements in the S-set, which, due to their endpoint c-values, cannot be made parallel to the new key link segment, are removed from the S-set. Since the slope of the key link segment is then to remain stationary, the c-values of each of its endpoints are then increased by one.

The second part of processing the S-set consists of actually going through each of the link segments in the S-set and adjusting them so that they are parallel to the key link segment. The manner in which a link segment is adjusted depends upon the c-value of its endpoints; thus, according to these c-values the link segment is adjusted in one of the steps from 13 to 17.

The algorithm is as follows; where r is an "angle of tolerance" to be set by the user:

- 1) Record all sets of parallel link segments. For all endpoints (nodes and bends), n , set $c(n) = 0$. Set all S-sets to null. Go to step 2.
- 2) Set $i = 1$; go to step 3.
- 3) If no link segments remain unplaced in S-sets go to step 5; otherwise, choose a link segment, n_i , not yet placed in an S-set, with preference for one to which others not yet placed are parallel, and

place the single element n_i in the S-set S_i ; mark link segment n_i as the key link segment for S_i and go to step 4.

4) Find a link segment, m , not yet placed in a set which meets the following conditions:

(i) If n_i is parallel to other link segments then m may be parallel only to n_i and its parallels, but to no others.

(ii) The slope of m measured over the first two quadrants is within r degrees of that of link segment n_i .

(iii) If m shares an endpoint with any other $p \in S_i$, the slopes of m and p must be opposite in sign with respect to that point.

If such a link segment is found, add it to S_i and go to step 4; otherwise, add one to i , and go to step 3.

5) For each set S_i : if the key link segment of S_i , n_i has parallels in S_i , form a set of the endpoints of n_i and of all link segments in S_i parallel to it; for each endpoint n in this set, add one to $c(n)$. In any case, all sets, S_i , are considered unprocessed at this point. Go to step 6.

6) If all sets, S_i , have been processed, or if the maximum number of slopes in all unprocessed sets is only one, terminate the algorithm; otherwise, choose the set S_i which contains the largest number of different slopes, and go to step 7.

7) Mark S_i as processed. If the key link segment for S_i , n_i has no parallels in S_i go to step 8; otherwise, if there are any

elements of S_i which are not parallel to n_i and whose endpoints, say a and b , are such that $c(a) > 1$ and $c(b) > 1$, remove these elements from S_i , and go to 10.

8) If there are any elements in S_i with endpoints a and b such that $c(a) > 1$ and $c(b) > 1$, choose one such, mark it as the key link segment for S_i (removing the mark from n_i), remove all other such from S_i , and go to step 9. Otherwise, choose an element of S_i , say j , for which the end points a_j and b_j are such that $c(a_j) + c(b_j) = \text{Max}(c(a_k) + c(b_k), k \in S_i)$. Mark j as the key link segment for S_i (removing this mark from n_i) and go to step 9.

9) Unless $|S_i| = 1$, for each of the endpoints a of the new key link segment for S_i , add one to $c(a)$. Go to step 10.

10) Set N equal to the key link segment for S_i , and remove N and all elements parallel to N from S_i . Go to step 11.

11) If $S_i = \emptyset$, go to step 6; otherwise, choose an element of S_i , call it n , and remove it from S_i . Go to step 12.

12) Record the positions of the endpoints of n , a and b , as A and B , respectively. If both endpoints of n have c -values of zero, go to step 13. If both have c -values of one, go to step 14. If one has c -value of one, and the other, of zero, go to step 15. If one has c -value of zero, go to step 16. Otherwise, go to step 17.

13) Rotate the link segment n around its centerpoint until it is parallel to N (a rotation of less than 180°), keeping its length constant.

Go to step 18.

14) Find the two link segments n_a and n_b ending on the two respective endpoints, a and b , for which these endpoints were assigned c -values of 1. It may be that either n_a or n_b consists of two parallel link segments from the same node, but extending in opposite directions. If n_a, n_b and N are parallel, go to step 11. If either n_a or n_b is parallel to N (say n_a is), and n_a is only a single link segment, then let c be the point of intersection of the extensions of n_a and n_b ; now, if the slope of n_a with respect to a is the same as that of a, c with respect to a , go to step 11, else, move b to coincide with c and go to step 18. If either n_a or n_b is parallel to N and the parallel segment is really two segments, go to step 11. Otherwise, find the midpoint of line segment n, c , and the line, Q , parallel to N which passes through c . The point at which Q intersects with the line based on the line segment n_a is the new position for a , and the intersection of Q with the extension of n_b is the new position for b . Go to step 18.

15) Let $c(a)$ be one, where a is an endpoint of n . Then let n_a be the link segment for which a was assigned the c -value of one. It may be that n_a consists of two parallel link segments from the same node but extending in opposite directions. If n_a is parallel to N , and n_a consists only of one link segment, reposition b so that the length of n remains constant, and so that n is parallel to N (rotating b through an arc of less than 180° around a). If the slopes of n_a and the resulting

n are opposite in sign with respect to a , go to step 18; otherwise, go to step 11. If n_a is parallel to N and consists of two link segments, go to step 11. In any other case, keeping the length of n constant, rotate b around a (through an arc of less than 180°) until n is parallel to N . Go to step 18.

16) Let a have a c -value of two or more. Keeping the length of n constant, rotate b around a (through an arc of less than 180°) until n is parallel to N . Go to step 18.

17) Let a have a c -value of two or more. Let n_b be the link segment for which b was assigned a c -value of one. It may be that n_b consists of two parallel link segments from the same node but extending in opposite directions. If n_b is parallel to N , go to step 11. Otherwise, move b along the extension of n_b until n is parallel to N . Go to step 18.

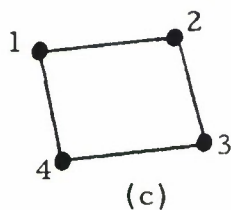
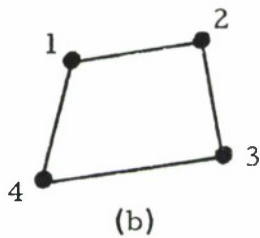
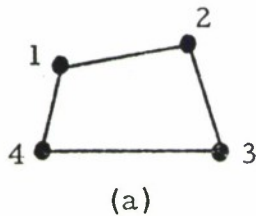
18) If any node or link overlaps have been caused by the movement of a or b , restore a and b to the old values A and B , respectively, and go to step 11. Otherwise, add one to $c(a)$ and $c(b)$ and go to step 11.

The algorithm is illustrated in figure A4-1, where r is about 45° . Relevant changes are noted after each step number. In the algorithm links which are parallel in the original layout remain parallel and fixed. The realization of this last condition however, causes some difficulty, in that two different sets of parallel links cannot be merged into one set.

If we are willing to give up the guarantee that parallel links remain parallel, we may remedy this by changing constraint (i) of step 4 to read:

(i) if both m and n are recorded as being parallel to other links, but m is not parallel to n , then we must remove the record of the membership of m in a set of parallel links (made in step 1) if we add m to S_i .

With this replacement, we have no guarantee that the result keeps originally parallel links parallel.



step:

1	$c(1) = c(2) = c(3) = c(4) = 0$
2, 3, 4	$i = 1$
	$S_1 = \{1-2, 3-4\}$
	$i = 2$
	$S_2 = \{2-3, 4-1\}$
5, 6	$S_i = S_1$
7, 8	$j = 1-2$
9	$c(1) = c(2) = 1$
10	$N = 1-2, S_1 = \{3-4\}$
11	$n = 3-4, S_1 = \emptyset$
12	$a = 3, b = 4$
	$A = p(3), B = p(4)$ in (a)
13	result shown in (b)
18	$c(3) = c(4) = 1$
6	$S_i = S_2$
7, 8	$j = 2-3$
9	$c(2) = c(3) = 2$
10	$N = 2-3, S_2 = \{4-1\}$
11	$n = 4-1, S_2 = \emptyset$
12	$a = 1, b = 4$
	$A = p(1), B = p(4)$ in (b)
14	$n_a = 1-2, n_b = 3-4$
	result shown in (c)
18	$c(1) = c(4) = 2$
11, 6	terminate

Figure A4-1

Appendix 5

DETAILS OF THE MOD SYSTEM

Most of the details of the system are straightforward and will not be reported on here. A few details of interest, however, will be discussed here. These include the internal representation of graph layouts, the basic structure of the system, and finally, the method used to determine which nodes and pins are indicated by the closed figures used for the move and copy operations.

INTERNAL REPRESENTATION

Three lists are used for representing graph layouts in the MOD system:

- 1) NLST - node and link storage - 7 word entries.
- 2) RSLST - relative line, shape, and line storage - 3 word entries.
- 3) PNTST - point storage - 3 word entries.

The representation of each entity in these lists is described below.

Each list is originally empty (entries are all zeroes). There is room for 50 NLST entries, 46 RSLST entries, and 50 PNTST entries. The entries are used as follows:

word bits

NODES

- | | | |
|---|------|--|
| 1 | 0-2 | 1 - indicates that this is a node |
| | 3-17 | pointer to <u>point</u> occupied by node |

2	0-17	pointer to <u>shape</u> of node
3	0-17	} 6 character name of node
4	0-17	
5	0-17	pointer to first outgoing <u>link</u> of node
6	0-17	pointer to first incoming <u>link</u> of node
7	0-17	(no permanent use)

LINK

1	0-2	2 - indicates that this is a link
	3-17	pointer to <u>node</u> at which link starts
2	0-17	pointer to <u>node</u> at which link ends
3	0-17	pointer to next <u>link</u> starting at node in word 1
4	0-17	pointer to next <u>link</u> ending at node in word 2
5	0-17	pointer to first <u>line</u> segment for link
6	0-17	pointer to ADP (<u>relin</u>) for link
7	0-17	pointer to EP (<u>relin</u>) for link

RELIN (relative line - used for ADP's, EP's, and shapes)

1	0	1 if this is a visible line of a shape; 0 otherwise
	1-2	1 - indicates that this is a relin
	3-17	pointer to next <u>relin</u> for shape line, next ADP, or next EP depending on what this relin is
2	3-17	} ΔX } for ADP's and EP's these are positions relative
3	3-17	
		ΔY } to node center; for shape lines this is x and y increment for this line

LINE (link segments only)

1	0-2	2 - indicates that this is a line
	3-17	pointer to <u>point</u> at which line starts
2	0-17	pointer to <u>point</u> at which line ends
3	0-17	pointer to next <u>line</u> in link

SHAPE

1	0-2	3 - indicates that this is a shape
	3-17	pointer to first line of shape as <u>relin</u>
2	0-2	first digit of function number for shape
	3-17	pointer to first ADP of shape as <u>relin</u>
3	0-2	second digit of function number for shape
	3-17	pointer to first EP of shape as <u>relin</u>

POINT

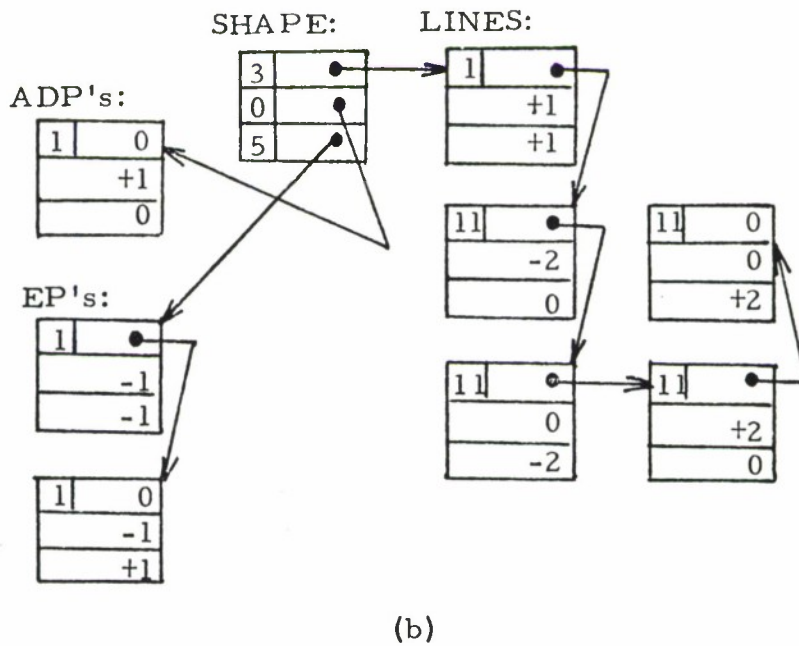
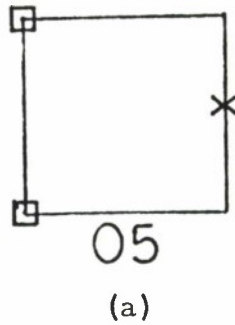
- | | | |
|---|------|--|
| 1 | 0-2 | 1 - indicates that this is a point |
| | 3-17 | pointer to <u>node</u> if this is a node position, or to the <u>line</u> which ends at this position if this is a bend point |
| 2 | 0-17 | x-coordinate of point |
| 3 | 0-17 | y-coordinate of point |

Note that where a word in an entry is not appropriate for a particular entity, the word is zero. For example, if a node has no outgoing links, the fifth word is zero, or, if an ADP is the last in a shape definition, bits 3-17 of word 1 are zero.

Thus, for example, the shape in figure A5-1a has the internal representation depicted in A5-1b. The node in figure A5-2a has the structure shown in an abbreviated manner in figure A5-2b. The lists NLST, RSLST, and PNTST reflect the current graph layout and defined shapes. When a new graph layout is read in on paper tape, the information is placed in these lists, and any previous information is destroyed. When an operation is initiated which would cause any of the three lists to be exceeded, the operation is aborted.

BASIC STRUCTURE

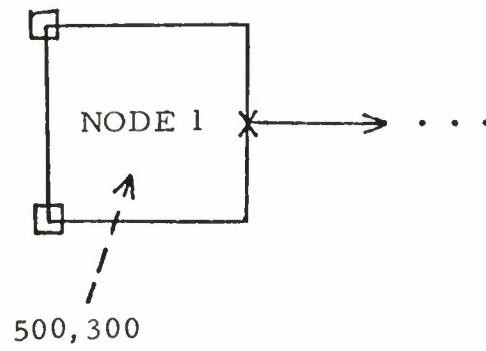
After initialization of any of the three systems, the control of the program is essentially determined by the interrupts received from the Rand tablet. A "main" program (MODMN for Input and Frame-maker, MODMN2 for Output) directs initialization and receives interrupts. Once interrupts are received, they are, in general, processed by an interrupt handling routine (LMNFRM for Input, LMNFR2 for



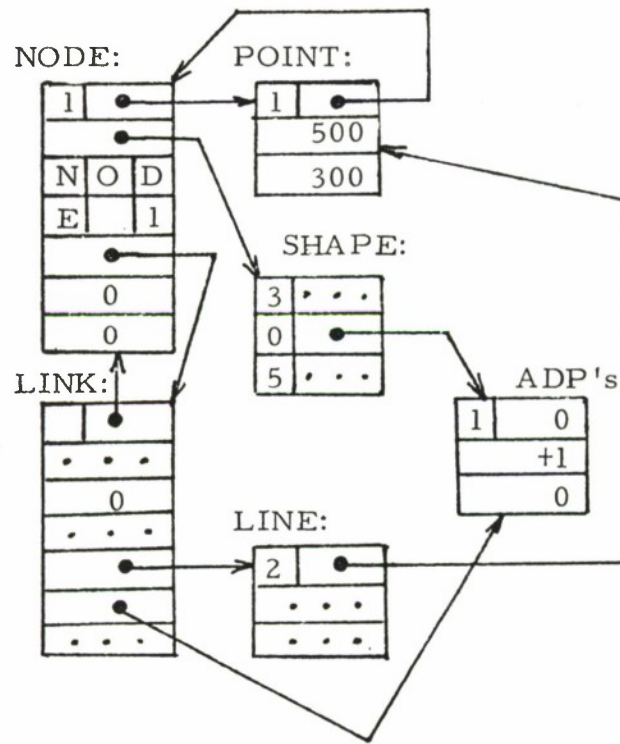
(b)

Figure A5-1

Output, and FMNFRM for Framemaker). These routines decode the type of action indicated by the pen movement and call routines to perform that action. For example, LMNFRM determines whether a label in the menu is being pointed at, and, if so, calls the routine to perform the action indicated by this label. LMNFRM also decides if part of the graph layout has been encircled. If so, it waits for the next pen movement, to decide whether it should move or copy what has been enclosed.



(a)



(b)

Figure A5-2

There are a few exceptions to this basic interrupt handling pattern. If, in Input or Framemaker, the DEFINE label is pointed at, a routine, GETDEF, is called which puts the define frame on the scope. For each subsequent interrupt received in MODMN, until "input" is pressed, interrupts are processed in GETDEF and appropriate action taken there. Once "input" is pressed, the normal mode of operation is continued. Similarly, in Framemaker, when "output" is pressed, FMNFRM calls a routine FOUTPT. FOUTPT then receives the next interrupt which is typewriter input, rather than tablet input, and takes the appropriate action. After one operation has been performed, however, the normal mode of operation is resumed.

Thus it is quite easy to see how additional layout modification algorithms might be added to MOD Output. The appropriate label must be added to those displayed in the menu of the main frame (this is done by adding a label to the permanent display file INFIL2). A routine must be written to perform the desired algorithm on the internal representation of the layout. And, instructions must be added to LMNFR2 to determine when this new label is being pointed at and, when it is, to call the routine to perform the algorithm. In the current implementation of MOD, unfortunately, due to the limited size of the PDP-1 used, little space remains for additional algorithms. A larger computer would be necessary for extension of the system.

METHOD FOR ENCLOSURE DETERMINATION

As a final note, a description is included of the method which is used to determine the contents of an enclosure made with the pen on part of the graph layout. This method is based on the following observation:

Given the interior region, A , defined by a closed curve, C , within a surface with boundary, B , a point, x , on the surface has the following properties:

- (1) If x lies in A , any line drawn from x to any point on B will cross C an odd number of times.
- (2) If x lies outside A , any line drawn from x to any point B will cross C an even number of times.

Observation (1) is supported by the fact that, if x is in A , for every crossing of C , the line passes alternately out of and into the region A . Since B is always outside A , the total number of crossings must be odd. A similar argument holds for (2).

Using this observation, then, once a closed curve (represented internally by a series of straight line segments) is drawn, we simply count, for each node and pin, x , the number of times a line from x to some point on the boundary of the main frame box crosses the enclosure. If the number is odd, x is in the enclosure and will be moved or copied; if the number is even, x remains unaffected.

REFERENCES

1. Allport, F. H., Theories of Perception and the Concept of Structure, Wiley, New York, 1955.
2. Anger, A. L., "An Algorithm for the Genus of a Graph," Ph.D. Thesis, Harvard University, October, 1965.
3. Attneave, F. and Arnoult, M. D., "The Quantitative Study of Shape and Pattern Perception," in Pattern Recognition, L. Uhr, Ed., Wiley, New York, 1966.
4. Baecker, R. M., "Planar Representation of Complex Graphs," Technical Note, 1967-1, Lincoln Laboratory, February, 1967.
5. Berge, C., The Theory of Graphs and Its Applications, Wiley, New York, 1962.
6. Birkhoff, G. D., Aesthetic Measure, Harvard University Press, Cambridge, 1933.
7. Boring, E. G., Sensation and Perception in the History of Experimental Psychology, Appleton-Century-Crofts, New York, 1942.
8. Breuer, M. A., "The Formulation of Some Allocation and Connection Problems as Integer Programs," Naval Research Logistics Quarterly, Volume 13, Number 1, March, 1966.
9. Breuer, M. A., "General Survey of Design Automation of Digital Computers," Proc. IEEE, Volume 54, December, 1966.
10. Busaker, R. G. and Saaty, T. L., Finite Graphs and Networks, McGraw Hill, New York, 1965.
11. Case, P. W., Graff, H. H., Griffith, L. E., Leclercq, A. R., Murley, W. B., and Spence, T. M., "Solid Logic Design Automation for IBM System/360," IBM J. Res. and Devel., Volume 8, April, 1964.
12. Di Giulio, H. A. and Tuan, P. L., "A Graph Manipulator for On-Line Network Picture Processing," AFIPS Conf. Proc., Volume 35, 1969.

13. Even, S., Lempel, A., and Cederbaum, I., "An Algorithm for Planarity Testing of Graphs," unpublished.
14. Fary, I., "On Straight Line Representation of Planar Graphs," *Acta Sci. Math.*, Volume 11, Number 4, 1948.
15. Flood, M. M., "The Traveling-Salesman Problem," *Operations Research*, 1956, pp. 61-75.
16. Gamblin, R. L., Jacobs, M. Q., and Tunis, C. J., "Automatic Packaging of Miniaturized Circuits," in *Advances in Electronic Circuit Packaging*, Volume 2, G. A. Walker, Ed., Plenum Press, New York, 1962.
17. Hake, H. W., "Form Discrimination and the Invariance of Form," in *Pattern Recognition*. L. Uhr, Ed., Wiley, New York, 1966.
18. Hartmann, G. W., *Gestalt Psychology*, Ronald Press, New York, 1935.
19. IBM, *System/360 Flowchart (360A-SE-22X)*, IBM Application Program, H20-0293-1.
20. Kalish, H. M., "Machine-Aided Preparation of Electrical Diagrams," *Bell Lab. Record*, Volume 41, Number 9, October 1963.
21. Katz, D., *Gestalt Psychology*, Ronald Press, New York, 1950.
22. Koffka, K., *Principles of Gestalt Psychology*, Routledge and Kegan Paul, London, 1935.
23. Kuhn, H. W., "The Hungarian Method for the Assignment Problem," *Naval Research Logistics Quarterly*, Volume 2, March-June, 1955.
24. Lee, C. Y., "An Algorithm for Path Connections and Its Applications," *IRE Trans. on Electronic Computers*, Volume 10, September, 1961.
25. Liu, C., *Introduction to Combinatorial Mathematics*, McGraw Hill, New York, 1968.

26. Mamelak, J. S., "The Placement of Computer Logic Modules," JACM, Volume 13, Number 4, October, 1966.
27. Michle, "Link-Length Minimization of Networks," Operations Research, Volume 6, 1958, pp. 232-243.
28. Miller, G. A., The Psychology of Communication, Penguin Books, Baltimore, 1967.
29. Mowatt, M. H., "Configurational Properties Considered 'Good' by Naive Subjects," in Readings in Perception, D. C. Beardslee and M. Wertheimer, Ed., Van Nostrand, Princeton, 1958.
30. Munkres, J., "Algorithms for the Assignment and Transportation Problems," J. SIAM, Volume 5, March, 1957.
31. Ore, O., Graphs and Their Uses, Random House, New York, 1963.
32. Rutman, R. A., "An Algorithm for Placement of Interconnected Elements Based on Minimum Wire Length," Proc. SJCC, 1964.
33. Steinberg, L., "The Backboard Wiring Problem, A Placement Algorithm," SIAM Rev., Volume 3, January, 1961.
34. Sutherland, I., "Computer Graphics - Ten Unsolved Problems," Datamation, May, 1966.
35. Thompson, D. W., On Growth and Form, Cambridge University Press, Cambridge, 1917.
36. Tiernan, J., "An Algorithm to Find the Elementary Circuits of a Graph," unpublished.
37. Vincent-Carrefaur, J. J., "Design Optimization of Small Logic Systems," Proc. 23rd Nat. Conf., ACM.
38. Wertheimer, M., "Principles of Perceptual Organization," in Readings in Perception, D. C. Beardslee and M. Wertheimer, Ed., Van Nostrand, Princeton, 1958.

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Harvard University Center for Research in Computing Technology Cambridge, Massachusetts 02138		2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	
		2b. GROUP N/A	
3. REPORT TITLE THE LAYOUT PROBLEM FOR GRAPHS			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) None			
5. AUTHOR(S) (First name, middle initial, last name) Martha Greenberg Dennis			
6. REPORT DATE August 1971		7a. TOTAL NO. OF PAGES 323	7b. NO. OF REFS 38
8a. CONTRACT OR GRANT NO. F19628-68-C-0379		9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-71-344	
b. PROJECT NO.			
c.		9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
d.			
10. DISTRIBUTION STATEMENT Approved for public release; distribution unlimited.			
11. SUPPLEMENTARY NOTES		12. SPONSORING MILITARY ACTIVITY Deputy for Command and Management Systems Hq Electronic Systems Division (AFSC) L G Hanscom Field, Bedford, Mass. 01730	
13. ABSTRACT The layout problem for graphs, the problem of automatically generating a representation of a graph on a two-dimensional surface, has been of interest in specific applications for many years, although little work has been done on the general problem. In this paper three approaches are taken towards solution of the problem. The first approach defines general layout qualities believed to be desirable. Means for measuring these qualities in layouts and algorithms for their realization are developed. A graph layout building and modification system is described which provides an experimental environment for such layout algorithms. The second approach considers layout from an application dependent point of view. A classification of layouts into types is developed according to application, and layout algorithms for each type are discussed. In this classification, a correlation is found between complexity of layout type and complexity of layout algorithm. An extension of the above graph layout building system is designed, which allows for inclusion of application dependent information in layout processing. The third approach, that of considering the layout of modifications of graphs, rather than layout of whole graphs, is briefly considered. It is concluded that this third approach is the least effective for finding solutions to the layout problem.			

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Graph layout Complexity of layout type Complexity of layout algorithm Application dependency						